UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Ingeniería del Software e Inteligencia Artificial

UNA APROXIMACIÓN ONTOLÓGICA AL MODELADO DE CONOCIMIENTO EN LOS DOMINIOS DE PLANIFICACIÓN.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Antonio A. Sánchez Ruiz-Granados

Bajo la dirección de los doctores

Pedro A. González Calero
Belén Díaz Agudo

Madrid, 2010

ISBN: 978-84-693-8795-5

Una aproximación ontológica al modelado de conocimiento en los dominios de planificación

TESIS DOCTORAL

Antonio A. Sánchez Ruiz-Granados

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Abril 2010

Documento maquetado con TEXIS v.1.0.

Una aproximación ontológica al modelado de conocimiento en los dominios de planificación

Memoria que presenta para optar al título de Doctor en Informática

Antonio A. Sánchez Ruiz-Granados

Dirigida por los Doctores

Pedro A. González Calero
Belén Díaz Agudo

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Abril 2010

Para mis padres

Agradecimientos

Da un poco de vértigo volver la vista atrás e intentar recordar a todas esas personas a las que, de un modo u otro, debo el haber llegado hasta aquí. Ha sido un largo camino, tan largo que aún ahora me parece mentira estar escribiendo estas lineas. Escribir una tesis es una experiencia única, una extraña mezcla de trabajo y afición que te inunda y toca irremediablemente a los que tienes más cerca. Durante los últimos años he tenido la suerte de poder dedicarme a lo que me gusta. He aprendido muchas cosas, he viajado a sitios a los que pensé que nunca iría y he conocido gente con inquietudes similares a las mías.

Y todo esto comenzó aquel día en el que decidí cometer la pequeña locura de dejar un buen trabajo, pero que no me llenaba, por buscar algo más... En realidad no sabía lo que buscaba pero tenía la intuición de que podía encontrarlo en la universidad, así que fui a hablar con Pedro, el profesor que me dirigió el proyecto fin de carrera. Pedro me ofreció una beca de 6 meses y la opción de participar de GAIA, nuestro entrañable grupo de investigación. Y desde entonces todo ha pasado muy rápido. Aunque si me paro a pensarlo, de eso hace ya unos cuantos años.

En primer lugar quiero dar las gracias a Pedro y Belén por ayudarme durante todo el proceso de gestación de esta tesis y por aguantar las prisas de última hora. Durante varias ocasiones a lo largo de estos años he creído llegar a caminos sin salida pero al final, entre todos, hemos conseguido sortear los obstáculos y llegar hasta aquí. Sin duda no lo habría conseguido sin ellos.

También quiero dar las gracias a mis compañeros de despacho (estén o no físicamente en él), con los que disfruto de interesantísimas charlas durante la comida y en los ratos de ocio. Ellos mejor que nadie comprenden mis alegrías y frustraciones porque han pasado o están pasando por lo mismo. Los momentos de distensión y los ánimos que he recibido en los momentos de más incertidumbre han sido fundamentales para volver a retomar el trabajo con energías renovadas. Gracias chicos.

No se me puede olvidar mi amigo Juan, compañero inseparable de aventuras y desventuras desde el instituto, que, aunque no entiende ni una palabra de lo que va esta tesis, siempre ha estado ahí cuando le he necesitado. También quiero dar las gracias a Lubov, mi amiga al otro lado del mundo, que

me ayudó muchísimo cuando estuve de estancia en Estados Unidos. Y por supuesto dar las gracias a Raquel, mi chica, que merece gran parte del mérito, porque las tesis se viven en pareja, sin duda. Ella ha estado siempre ahí, apoyándome y dándome ánimos, aguantando los momentos malos y compartiendo los buenos, y metiéndome prisa para que la acabara de una vez. Sin ella aún estaría escribiendo la introducción.

Y para el final dejo a los más importantes, mi familia. Y en especial a mis padres que siempre me han apoyado de manera incondicional y han hecho todo lo posible para ayudarme a alcanzar mis sueños. Si alguien se merece esta tesis son ellos. Así que aquí está, os la dedico.

Resumen

Históricamente la comunidad de planificación ha concentrado sus esfuerzos en la creación de potentes algoritmos de búsqueda para resolver problemas en dominios muy sencillos en los que se usa muy poco conocimiento. Sin embargo, cuando intentamos utilizar técnicas de planificación para resolver problemas del mundo real nos encontrarnos con dominios ricos en conocimiento, difíciles de modelar y con gran cantidad de información asociada. En este tipo de dominios surgen nuevos problemas relacionados con la adquisición y gestión de grandes bases de conocimiento.

Nuestra propuesta consiste en utilizar ontologías para modelar el conocimiento de carácter estático asociado a estos dominios. En concreto, nos centraremos en un tipo de ontologías con una base formal bien fundada (las lógicas descriptivas) que ofrecen un buen compromiso entre expresividad y complejidad computacional. Además, estas ontologías son uno de los pilares fundamentales de la web semántica, por lo que existe toda la infraestructura montada a su alrededor de la que podemos aprovecharnos: lenguajes estándar, editores visuales, sistemas de razonamiento, etc. Uno de los objetivos de este trabajo consiste en *estudiar qué pueden aportar las ontologías y toda esta infraestructura montada a su alrededor al modelado del conocimiento asociado a los dominios de planificación.*

Por otra parte, las ontologías proporcionan un vocabulario rico con el que poder expresar conocimiento usando distintos niveles de detalle. Esta capacidad de razonamiento abstracto permite realizar inferencias interesantes a partir del conocimiento disponible. Además, permite plantear problemas de planificación abstractos cuyas soluciones pueden aplicarse a multitud de problemas concretos, algo que resulta especialmente interesante en aproximaciones basadas en casos. En este trabajo también *investigaremos cómo podemos aprovechar la capacidad de razonamiento a distintos niveles de abstracción de las lógicas descriptivas.*

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

En este primer capítulo describimos las ideas que motivan el desarrollo de esta tesis, explicamos de manera breve las propuestas que desarrollaremos a lo largo de la memoria, y enumeramos las contribuciones de nuestro trabajo. El capítulo finaliza explicando la estructura de la memoria y las publicaciones a las que ha dado lugar.

1.1. Motivación

Históricamente la comunidad de planificación ha concentrado sus esfuerzos en la creación de potentes algoritmos de búsqueda para resolver problemas en dominios muy sencillos. Estos dominios suelen consistir en pequeños juegos de tipo lógico descritos con lenguajes poco expresivos de tipo proposicional. Sin embargo, cuando intentamos aplicar técnicas de planificación a problemas reales nos encontramos con dominios ricos en conocimiento, dominios difíciles de modelar con mucha información asociada. En estos escenarios surgen nuevos problemas relacionados con la adquisición, mantenimiento y explotación de grandes bases de conocimiento, problemas que ya han empezado a atraer la atención de la comunidad de planificación. Prueba de ello son los distintos talleres (KEPS) y competiciones sobre Ingeniería del Conocimiento y Planificación (ICKEPS) celebrados en los últimos años [40, 11, 19, 120, 12].

Adquirir el conocimiento necesario para modelar un dominio de manera formal es un proceso muy complejo y costoso. Pero las bases de conocimiento no sólo hay que crearlas y verificarlas, también hay que mantenerlas. Estamos ante un problema tan importante que existe toda una disciplina, la Ingeniería del Conocimiento, enfocada a mejorar los procesos de adquisición y gestión de conocimiento. Una de las ideas básicas para facilitar el modelado de dominios complejos, y por tanto abaratar costes, consiste en fomentar que se comparta y reutilice el conocimiento ya adquirido. Surge de este modo el concepto de *ontologías* [48]: artefactos semánticos que describen de

forma explícita la estructura de un dominio mediante jerarquías conceptuales. Las ontologías fomentan que el conocimiento se comparta entre distintos sistemas representando la terminología de un dominio de manera intuitiva y extensible.

No en vano las ontologías son un pilar esencial para representar conocimiento en la web semántica [16]. Este ambicioso proyecto pretende extender la web actual con etiquetas semánticas y, de ese modo, facilitar la interoperabilidad entre sistemas informáticos y usuarios. Debido a la cantidad y variedad del conocimiento disponible en la web, esta comunidad ha prestado especial atención a los problemas relativos a la adquisición y gestión de conocimiento. Como resultado de todo este esfuerzo han surgido lenguajes estándar para representar ontologías [59], editores visuales [76], razonadores muy optimizados que permiten depurar y gestionar grandes cantidades de información de manera eficiente [126, 140, 52], repositorios de ontologías [65], etc.

Ante esta situación parece natural preguntarse si es posible aprovechar todo este esfuerzo para mejorar los procesos de adquisición y gestión del conocimiento relacionado con los dominios de planificación. Esta es una de las ideas básicas de este trabajo, *estudiar qué pueden aportar las ontologías y toda la infraestructura montada a su alrededor al modelado del conocimiento asociado a los dominios de planificación.*

Por otra parte, las ontologías también permiten describir situaciones con distinto nivel de detalle. En una jerarquía conceptual los conceptos de la parte superior, que describen categorías abstractas, se van especializando a medida que descendemos por la jerarquía, hasta alcanzar los conceptos en las hojas que se usan para representar tipos concretos de entidades. Disponer de toda esta riqueza en el vocabulario nos permite describir la información y razonar sobre ella a distintos niveles de abstracción. Podemos utilizar esta interesante característica de las ontologías para: (1) *mejorar la interactividad entre los sistemas de planificación y los usuarios*; y (2) *fomentar la reutilización de planes una vez generados.*

Finalmente, existen distintos lenguajes estándar en la web semántica para representar ontologías dependiendo de la expresividad y garantías computacionales que se necesiten. Nosotros nos centraremos en el uso de ontologías OWL-DL [59], un lenguaje respaldado por una familia de lenguajes formales bien estudiados, las lógicas descriptivas (DLs, del inglés *Description Logics*) [6], que proporcionan un buen compromiso entre expresividad y complejidad computacional.

1.2. Propuesta

Nuestra propuesta consiste en usar ontologías OWL-DL para representar el conocimiento de tipo estático que aparece en los dominios de plani-

ficación. Esta aproximación nos permite aprovechar toda la infraestructura desarrollada en la comunidad de la web semántica para adquirir y gestionar conocimiento:

- La existencia de *lenguajes estándar* y *repositorios de ontologías* facilita el modelado de dominios, y permite aprovechar el esfuerzo que otros han invertido para representar dominios similares.
- Los *editores de ontologías visuales* proporcionan entornos amigables que simplifican el acceso a estas tecnologías. Además, permiten gestionar grandes proyectos que, debido a su complejidad, serían inviables sin estas herramientas.
- Los *razonadores* existentes nos permitirán *detectar y corregir inconsistencias* en la definición de dominios y problemas de planificación. Estos razonadores se integran en los entornos de edición de ontologías para facilitar su uso.
- Finalmente, los razonadores están muy optimizados y son capaces de *gestionar grandes cantidades de información de manera eficiente*. Nuestra aproximación utiliza las características avanzadas de razonamiento incremental de uno de estos sistemas para *realizar inferencias interesantes durante el proceso de búsqueda*.

Representar el conocimiento estático de un dominio de planificación usando ontologías parece tener numerosas ventajas. Sin embargo, todo este esfuerzo resulta inútil si no disponemos de un planificador capaz de gestionar estos modelos. Surgen, por tanto, dos alternativas: traducir las ontologías al lenguaje estándar de planificación, PDDL [46], y usar un planificador convencional; o construir un planificador basado en lógicas descriptivas. Como veremos a continuación, ambas alternativas tienen ventajas e inconvenientes.

La ventaja de usar un planificador estándar es que podemos aprovechar los potentes algoritmos de búsqueda que ya existen para resolver problemas de planificación. El principal inconveniente se esconde en el proceso de traducción del conocimiento del dominio, ya que las posibilidades que ofrece PDDL para representar conocimiento ontológico son muy limitadas. Para realizar esta traducción es necesario que un experto en las dos áreas detecte y extraiga de las ontologías todo el conocimiento necesario para resolver los problemas de planificación, y lo represente de forma adecuada usando el lenguaje del planificador. Este proceso no sólo es difícil y costoso, también conlleva los problemas relativos a mantener sincronizado el conocimiento representado en dos lenguajes distintos.

La segunda alternativa, por la que nos hemos decantado en este trabajo, consiste en construir un planificador que gestione el conocimiento representado en ontologías usando un razonador de DLs. De este modo podemos

explotar el conocimiento del dominio para realizar interesantes inferencias durante el proceso de búsqueda. La principal desventaja de esta aproximación es que los tiempos necesarios para resolver problemas serán mucho mayores, ya que, hoy por hoy, no disponemos de heurísticas inteligentes para recorrer el espacio de búsqueda. Aún así, pensamos que este enfoque es interesante en dominios donde la dificultad no depende tanto de la longitud de los planes solución como de la gestión del conocimiento disponible.

Por otra parte, proponemos aumentar la aplicabilidad de los planes generados aprovechando la capacidad de razonamiento con información incompleta de las DLs y la riqueza expresiva de las ontologías para representar conocimiento a distintos niveles de abstracción. Esta idea facilita el uso de técnicas de planificación basada en casos (CBP, del inglés *Case-Based Planning*) [57] en las que las soluciones se construyen adaptando soluciones que funcionaron anteriormente en problemas similares. En concreto, nosotros describiremos una *modelo de planificación basado en casos que saca partido a las capacidades de inferencia de un razonador DL para reutilizar planes generados anteriormente*. Además, el uso de casos puede ayudar a aliviar los problemas de rendimiento del planificador generativo, evitando tener que volver a recorrer gran parte del espacio de búsqueda para resolver problemas similares a los ya resueltos.

Finalmente, mostraremos la viabilidad de nuestra propuesta con dos ejemplos concretos relacionados con el mundo de los videojuegos. En ambos casos estamos ante dominios con mucho conocimiento asociado en los que resulta natural el uso de ontologías. En el primer ejemplo utilizaremos planificación en un videojuego educativo para personalizar los ejercicios que se proponen a cada estudiante, adaptándolos a sus necesidades concretas. En el segundo ejemplo usaremos planificación en la fase de creación de un videojuego para ayudar a los diseñadores a definir el comportamiento de los personajes no jugadores (NPCs, del inglés *Non Player Characters*).

1.3. Contribuciones

A continuación enumeramos las contribuciones concretas de este trabajo:

- En primer lugar proponemos el uso de ontologías OWL-DL para modelar dominios de planificación con una estructura y vocabulario complejos. De este modo podremos aprovechar la infraestructura desarrollada alrededor de la web semántica para adquirir y gestionar conocimiento: lenguajes estándar, editores de ontologías, potentes razonadores, repositorios, base formal de las DLs, etc.
- También estudiamos las limitaciones de PDDL para representar dominios ricos en conocimiento y, como consecuencia, los problemas que surgen al intentar traducir el conocimiento de las ontologías al lenguaje

estándar de planificación. Aún así, proponemos un esquema de traducción que permite transcribir *parcialmente* el conocimiento representado en las ontologías.

- Después recopilamos, a partir de varios trabajos previos, un modelo teórico de planificación basado en DLs. Abordamos tanto el modelo de planificación clásico, basado en el recorrido del espacio de estados, como un modelo jerárquico, basado en el recorrido del espacio de descomposición de tareas. Para finalizar, resumimos el estado actual de la investigación en este área y las limitaciones de carácter práctico que encontramos.

- Como parte del desarrollo de esta investigación hemos creado el sistema DLPlan, un planificador que representa el conocimiento usando ontologías OWL-DL y que utiliza una aproximación sintáctica a la ejecución de acciones. Describiremos la arquitectura del sistema, en la que la gestión del conocimiento estático se delega en el razonador Pellet [126], y mostraremos varios ejemplos de uso.

- Además, describimos un modelo teórico de planificación basada en casos que aprovecha la capacidad de razonamiento con información incompleta de las DLs y la riqueza expresiva de las ontologías para aumentar la aplicabilidad de los planes. La fase de adaptación de los casos se basa en las inferencias que un razonador es capaz de realizar a partir de la ontología que describe el dominio.

- Finalmente, mostramos dos casos de uso relacionados con el mundo de los videojuegos: (1) adaptación automática de ejercicios en un videojuego educativo para adecuarlos al perfil del estudiante, y (2) ayuda a los diseñadores a la hora de construir los árboles de comportamiento que posteriormente dirigirán a los NPCs del juego.

1.4. Estructura de la memoria

Capítulo 2. Web semántica, ontologías y planificación.

En este capítulo repasamos las tecnologías y herramientas que utilizamos en el resto de la memoria. Comenzamos revisando el uso de ontologías para representar y compartir conocimiento, y la infraestructura desarrollada en la web semántica para facilitar su uso: lenguajes estándar, lógicas descriptivas, editores de ontologías, razonadores, etc. Más tarde introducimos el problema de la planificación y resumimos tres aproximaciones que usaremos más adelante: búsqueda en el espacio de estados, planificación jerárquica y planificación basada en casos. A continuación describimos PDDL, el lenguaje estándar de planificación, y los cambios que ha sufrido a lo largo de los años.

El capítulo termina con una recopilación de los trabajos más significativos relativos al uso de ontologías para modelar conocimiento de planificación.

Capítulo 3. Dominios de planificación ricos en conocimiento.

En este capítulo comenzamos distinguiendo dos tipos de conocimiento que aparecen en los dominios de planificación: estático y dinámico. Después repasamos los dominios de la última competición de planificación para concluir que esta comunidad ha concentrado sus esfuerzos en problemas cuya dificultad se basa en la longitud de los planes solución y no en la gestión del conocimiento de tipo estático. A continuación proponemos el uso de ontologías OWL-DL para modelar dominios en los que es necesario explotar grandes bases de conocimiento. Finalmente, estudiamos los problemas que surgen cuando intentamos traducir este tipo de dominios a PDDL, un lenguaje que no fue diseñado con este propósito.

Capítulo 4. Planificación usando lógicas descriptivas.

En este capítulo introducimos los conceptos básicos necesarios para definir un modelo de planificación que utiliza DLs para representar el conocimiento. En concreto, describimos dos enfoques distintos al problema: planificación como búsqueda en el espacio de estados y planificación como descomposición de redes jerárquicas de tareas (HTN, del inglés *Hierarchical Task Network*). Terminamos el capítulo describiendo el estado actual de la investigación en este área y los problemas de carácter práctico que existen aún.

Capítulo 5. DLPlan: un planificador basado en OWL-DL.

Como parte de este trabajo hemos desarrollado un planificador al que hemos llamado DLPlan que, usando las ideas del capítulo anterior, implementa un modelo simplificado donde los efectos de las acciones se interpretan de manera sintáctica. En este capítulo describimos la arquitectura de dicho sistema, mostramos varios ejemplos de uso, y hacemos algunos experimentos para comprobar su rendimiento.

Capítulo 6. Planificación basada en casos y ontologías.

En este capítulo describimos un modelo de planificación basado en casos que aprovecha la capacidad de razonamiento con información incompleta de las DLs y la riqueza expresiva de las ontologías para representar conocimiento a distintos niveles de abstracción. En concreto describimos cómo podemos generalizar las condiciones iniciales de un plan para aumentar su aplicabilidad, y cómo podemos valernos de operaciones de subsunción para

recuperar y adaptar planes usando únicamente el conocimiento de las ontologías del dominio. Después introducimos jCOLIBRI [111], una plataforma desarrollada en nuestro grupo de investigación para la creación de sistemas CBR [2], y describimos cómo podríamos integrar este tipo de planificador en la plataforma.

Capítulo 7. De la teoría a la práctica.

En este capítulo aplicamos las ideas de este trabajo en dos dominios relacionados con los videojuegos. En el primer caso usamos *Javy* [49, 51], un videojuego educativo que enseña conceptos relacionados con la compilación y ejecución de programas Java. En este caso usamos planificación para personalizar una base de ejercicios, creada de antemano por expertos, a las necesidades específicas de cada estudiante. En el segundo ejemplo explicamos cómo podemos usar planificación, durante la fase de desarrollo de un videojuego, para ayudar a los diseñadores en la creación de árboles de comportamiento [109, 110]. Estos árboles de comportamiento se utilizan para definir cómo deben reaccionar los personajes no jugadores ante distintas situaciones que se pueden dar en el juego.

Capítulo 8. Conclusiones y trabajo futuro.

En el último capítulo resumimos el trabajo realizado, presentamos las conclusiones más importantes que hemos aprendido, y proponemos posibles líneas de trabajo futuro.

1.5. Publicaciones

Este trabajo de investigación ha dado lugar a las siguientes publicaciones:

1. Sánchez-Ruiz, A. A., Llansó, D., Gómez-Martín, M. A. y González-Calero, P. A. Authoring Behaviour for Characters in Games Reusing Abstracted Plan Traces. En *IVA* (editado por Z. Ruttkay, M. Kipp, A. Nijholt y H. H. Vilhjálmsson), vol. 5773 de *Lecture Notes in Computer Science*, páginas 56–62. Springer, 2009. ISBN 978-3-642-04379-6.

2. Sánchez-Ruiz, A. A., González-Calero, P. A. y Díaz-Agudo, B. Abstraction in Knowledge-Rich Models for Case-Based Planning. En *ICCBR* (editado por L. McGinty y D. C. Wilson), vol. 5650 de *Lecture Notes in Computer Science* , páginas 313–327. Springer, 2009. ISBN 978-3-642-02997-4.

3. Sánchez-Ruiz, A. A., Llansó, D., Gómez-Martín, M. A. y González-Calero, P. A. Authoring Behaviours for Game Characters

Reusing Automatically Generated Abstract Cases. En *ICCBR Workshop* (editado por S. J. Delany), páginas 129–137. 2009.

4. Sánchez-Ruiz, A. A., Gómez-Martín, P. P., Díaz-Agudo, B. y González-Calero, P. A. Adaptation through Planning in Knowledge Intensive CBR. En *ECCBR* (editado por K.-D. Althoff, R. Bergmann, M. Minor y A. Hanft), vol. 5239 de *Lecture Notes in Computer Science*, páginas 503–517, Springer, 2008. ISBN 978-3-540-85501-9.

5. Sánchez-Ruiz, A. A., González-Calero, P. A. y Díaz-Agudo, B. Combining HTN-DL Planning and CBR to Compound Semantic Web Services. En *Proceedings. of the KWEPSY2007: Knowledge Web PhD Symposium* (editado por E. P. B. Simperl, J. Diederich y G. Schreiber), vol. 275 de *CEUR Workshop Proceedings*, páginas 104–105. CEUR-WS.org, 2007.

6. Sánchez-Ruiz, A. A., González-Calero, P. A. y Díaz-Agudo, B. Planning with Description Logics and Syntactic Updates. En *Planning, Scheduling and Constraint Satisfaction (CAEPIA 2007 Workshop)* (editado por M. A. Salido y J. Fdez-Olivares), páginas 140–150. Universidad de Salamanca, 2007.

7. Sánchez-Ruiz, A. A., Lee-Urban, S., Muñoz-Avila, H., González-Calero, P. A. y Díaz-Agudo, B. Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval. En *Proceedings of the ICAPS-07 Workshop on Planning in Games*. 2007.

8. Díaz-Agudo, B., González-Calero, P. A., Recio-García, J. A. y Sánchez-Ruiz, A. A. Building CBR systems with jCOLIBRI. *Special Issue on Experimental Software and Toolkits of the Journal Science of Computer Programming*, vol. 69(1-3), páginas 68–75, 2007. ISSN 0167-6423.

9. Sánchez-Ruiz, A. A., Recio-García, J. A., Díaz-Agudo, B. y González-Calero, P. A. Case Structures in jCOLIBRI. *Expert Update* , vol. 9(2), 2006. ISSN 1465-4091.

10. Sánchez-Ruiz, A. A., Recio-García, J. A., González-Calero, P. A. y Díaz-Agudo, B. Towards Semi-Automatic Composition of CBR Systems in jCOLIBRI. En *Proccedings of the 11th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 48–59. CMS Press, University of Greenwich, 2006. ISBN 987-1-904521-39-6.

11. Recio-García, J. A., Díaz-Agudo, B., González-Calero, P. A. y Sánchez-Ruiz, A. A. Ontology based CBR with jCOLIBRI. En *Applications and Innovations in Intelligent Systems XIV. Proceedings of*

AI-2006, the Twenty-sixth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (editado por R. Ellis, T. Allen y A. Tuson), páginas 149–162. Springer, Cambridge, United Kingdom, 2006. ISBN 1-84628-665-4.

12. Recio-García, J. A., Díaz-Agudo, B., Sánchez-Ruiz, A. A. y González-Calero, P. A. Lessons Learnt in the Development of a CBR Framework. En *Proccedings of the 11th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 60–71. CMS Press, University of Greenwich, 2006. ISBN 987-1-904521-39-6.

13. Recio-García, J. A., Sánchez-Ruiz, A. A., Díaz-Agudo, B. y González-Calero, P. A. jCOLIBRI 1.0 in a nutshell. A software tool for designing CBR systems. *En Proccedings of the 10th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 20–28. CMS Press, University of Greenwich, 2005. ISBN 1-904521-30-4.

Capítulo 2

Web semántica, ontologías y planificación

En este capítulo introducimos las tecnologías y herramientas que usaremos en el resto de la memoria. El capítulo está dividido en dos partes. En la primera nos centramos en la idea de *ontología* [48] como mecanismo estándar para representar y compartir conocimiento, y describimos toda la infraestructura desarrollada en la web semántica alrededor de este concepto. Comenzaremos distinguiendo distintos tipos de ontologías según su estructura y contenido, y veremos los lenguajes que se utilizan para representarlas. Después nos centraremos en una familia de lenguajes formales, las lógicas descriptivas [6], que permiten describir un tipo de ontologías de vital importancia para este trabajo. Finalmente, enumeraremos algunas de las herramientas que nos ayudarán a gestionar este tipo de conocimiento como editores visuales y sistemas de razonamiento.

La segunda parte de este capítulo la dedicamos a la planificación. Comenzaremos con una pequeña introducción al problema y después describiremos tres aproximaciones que usaremos más adelante en este trabajo: búsqueda en el espacio de estados, planificación jerárquica y planificación basada en casos. A continuación estudiaremos el lenguaje estándar de planificación PDDL [46] y veremos los cambios que ha sufrido a lo largo de los años. Para finalizar el capítulo, recopilamos los trabajos más significativos relativos al uso de ontologías para modelar conocimiento de planificación.

2.1. Web semántica y ontologías

La *web semántica* [16] es un ambicioso proyecto que trata de extender la *web* actual añadiendo información semántica que los ordenadores puedan procesar de manera automática. Disponer de información *bien definida* a nivel semántico permitiría mejorar la interoperabilidad entre sistemas y

usuarios, permitiendo ofrecer mejores servicios.

La web actual está compuesta por millones de documentos HTML en los que la información se describe mediante texto y contenidos multimedia (imágenes, audio y vídeo). Esta manera de representar información es muy intuitiva para los seres humanos pero no resulta adecuada para las máquinas, que trabajan mejor con bases de datos donde la información está muy estructurada. La web semántica propone describir los datos y sus relaciones mediante etiquetas semánticas, usando estándares respaldados por lenguajes formales cuyas propiedades de expresividad y razonamiento son bien conocidas.

Describir el conocimiento a nivel semántico es una labor costosa y complicada, así que resulta imprescindible disponer de tecnologías y herramientas que potencien la posibilidad de compartir y reutilizar dicho conocimiento. Uno de los principales mecanismos propuestos para aliviar este problema es el uso de *ontologías*. Las ontologías permiten representar el conocimiento de manera intuitiva y extensible mediante el uso de jerarquías conceptuales. La comunidad de la web semántica ha invertido mucho esfuerzo para facilitar su creación y mantenimiento, desarrollando lenguajes de representación estándar, editores de ontologías, razonadores, repositorios, etc.

En los siguientes apartados profundizaremos en el concepto de ontología que se utiliza en la web semántica y en la infraestructura que se ha desarrollado alrededor de este concepto para facilitar la adquisición y gestión del conocimiento.

2.1.1. Ontologías

La definición más extendida de ontología es la presentada por Gruber [48]: *"una ontología es una especificación explícita de una conceptualización"*. Entendemos por conceptualización una representación abstracta y simplificada de un fragmento de la realidad. Las ontologías, por tanto, son estructuras semánticas que codifican de manera explícita la estructura de un dominio, valiéndose para ello de objetos, conceptos y otras entidades, así como las relaciones que existen entre ellos.

El término *ontología* proviene originalmente de la filosofía. La ontología filosófica se ocupa de la definición del *Ser* y de establecer las categorías fundamentales o modos generales de ser de las cosas. En los sistemas basados en conocimiento lo que existe es exactamente lo que puede ser representado, y el término ontología se toma prestado para denotar una representación declarativa y consensuada del vocabulario de un determinado dominio.

Los elementos básicos de una ontología son los *objetos*, sus *propiedades* y las categorías o *clases* a las que pertenecen. Las entidades se describen mediante su relación con otras entidades, siendo las dos relaciones más comunes *is-a* y *part-of*. La relación *is-a* permite definir jerarquías conceptuales

donde los conceptos más abstractos, situados en la parte superior, se van especializando sucesivamente hasta alcanzar conceptos concretos en la parte inferior de la jerarquía. La relación *part-of* permite definir entidades complejas a partir de otras entidades más sencillas de manera incremental. Además, los nombres de las entidades suelen venir acompañados de descripciones de texto para facilitar su comprensión a los usuarios, y de axiomas formales que limitan la interpretación y uso correcto de los términos.

Podemos clasificar las ontologías usando dos dimensiones [60]: (1) la cantidad y tipo de estructura de la conceptualización; y (2) el tipo de conocimiento que se representa. Con respecto a la primera dimensión distinguimos 3 categorías:

- *Ontologías terminológicas de tipo lexicón*: describen los términos necesarios para representar conocimiento en el dominio del discurso. Un ejemplo de este tipo de ontologías es [79] en el campo de la medicina.
- *Ontologías de información*: especifican la estructura de bases de datos. Un ejemplo sería el nivel 1 del proyecto PEN & PAD [118], un armazón para modelar los expedientes médicos de pacientes.
- *Ontologías de modelado del conocimiento*: describen la información usando conceptualizaciones con una estructura interna mucho más rica que las ontologías de información. Este tipo de ontologías suelen representar el conocimiento teniendo en cuenta el uso que se le va a dar. El nivel 2 del modelo PEN & PAD sería un ejemplo de este tipo de ontologías.

Desde el punto de vista del tipo de conocimiento representado, distinguimos 4 categorías:

- *Ontologías de la aplicación*: contienen las definiciones necesarias para modelar el conocimiento requerido en una aplicación particular. Este tipo de ontologías no son directamente reutilizables, contienen conceptos extraídos de ontologías generales y de dominios particulares que son adaptados para la aplicación específica. A veces estas ontologías también incorporan extensiones para representar tareas y métodos de resolución.
- *Ontologías de dominio*: describen conceptualizaciones de dominios concretos. El objetivo de estas ontologías es capturar, de manera reutilizable, el conocimiento relativo a un área o campo determinado.
- *Ontologías generales*: este tipo de ontologías definen conceptos abstractos comunes a distintas áreas, estableciendo un marco común y uniforme para integrar conocimiento de distintos dominios. De hecho, los

conceptos de las ontologías de dominio suelen definirse como especializaciones de estos conceptos generales. Algunos ejemplos de conceptos que solemos encontrar en estas ontologías son *estado*, *evento*, *proceso*, *acción*, *componente*, etc. La línea que separa las ontologías de dominio y las generales no es siempre clara, por lo que debemos llegar a consenso sobre dónde situar los conceptos frontera.

- *Ontologías de representación*: describen la conceptualización que subyace a los formalismos de representación [31]. Proporcionan las primitivas necesarias para representar conocimiento en ontologías generales y de dominio de manera neutra, es decir, sin tener en cuenta el tipo de conocimiento que estas últimas van a representar.

Desde el punto de vista de la representación, las ontologías se pueden representar en una amplia variedad de formalismos, dependiendo siempre de la expresividad requerida y de cómo se vaya a utilizar dicho conocimiento. En este trabajo nosotros nos centraremos en las *ontologías de modelado de conocimiento para dominios concretos*, y las representaremos usando OWL-DL [59], un lenguaje estándar de la web semántica que proporciona un buen compromiso entre expresividad y complejidad computacional.

2.1.2. Tecnologías y lenguajes estándar en la Web

La W3C (*World Wide Web Consortium*) ha propuesto un conjunto tecnologías estándar para la representación de la información en la web semántica. La figura 2.1 muestra un pequeño resumen de las tecnologías más significativas representadas en forma de pila. La arquitectura básica que se propone está compuesta por varias capas conceptuales: la capa inferior describe la información a nivel sintáctico; la siguiente capa añade descripciones semánticas usando, por ejemplo, *ontologías* y *reglas*; a continuación vendrían los sistemas de inferencia o *razonadores* que gestionan la información semántica a nivel lógico, ofreciendo distintos servicios de razonamiento; y finalmente, en la capa superior, estarían las aplicaciones semánticas.

A continuación damos un rápido repaso a las principales tecnologías implicadas en este esquema.

URI / IRI

Los distintos lenguajes usados en la Web Semántica permiten describir entidades en la Web. Los URI (*Uniform Resource Indentifier*) [15] son una manera estándar de identificar entidades mediante cadenas de texto. Los IRI (*Internationalized Resource Identifier*) [33] son similares a los URI pero permiten utilizar caracteres de otras lenguas distintas de la inglesa.

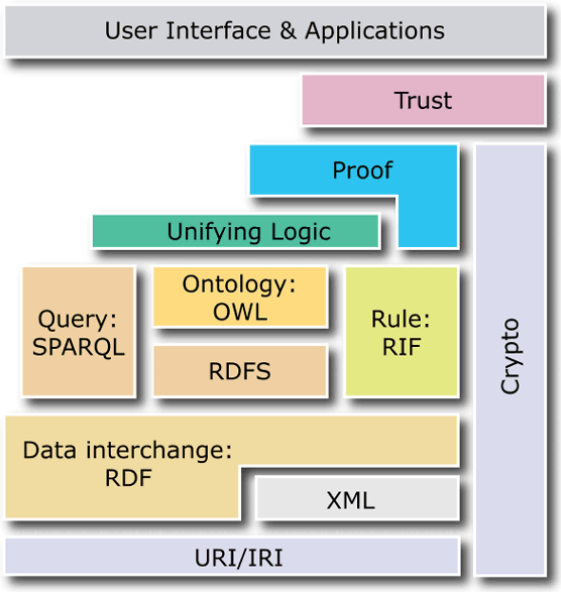


Figura 2.1: Arquitectura de la web semántica (Tim Berners-Lee)

XML / XML Schema

XML (*Extensible Markup Language*) [106] es un lenguaje de marcado de propósito general que permite a cada usuario definir sus propias etiquetas. Permite representar y compartir información estructurada de manera estándar, facilitando la comunicación entre distintas aplicaciones. Existen cientos de lenguajes basados en XML para representar información en distintos dominios (XHTML, RSS, MathML, SVL, ...), y cada uno de ellos define sus propias etiquetas dotándolas de una semántica particular. En particular, los lenguajes de la web semántica que describiremos a continuación están basados en XML.

XML Schema [39] permite definir distintos tipos de documentos XML estableciendo restricciones sobre su estructura y contenido. Los documentos XML que cumplan las restricciones impuestas por el esquema se denominan *válidos*.

RDF / RDFS

RDF (*Resource Description Framework*) [13] es un lenguaje de propósito general para representar información en la web. Permite expresar información usando tripletes con la forma *(sujeto, predicado, objeto)*. RDF permite definir modelos de datos en forma de grafo dirigido donde cada triplete define un nodo origen (sujeto), una arista (predicado) y un nodo destino (objeto).

Los nodos del grafo pueden contener una URI que identifique una entidad concreta, un literal (cadenas, números, ...), o bien un nodo en blanco. Las aristas se representan mediante URIs que establecen relaciones entre nodos o bien definen pares atributo-valor.

RDFS (*RDF Schema*) [25] es una extensión semántica de RDF que proporciona los elementos básicos para describir ontologías: *clases*, *propiedades* e *instancias*. Las clases representan conjuntos de elementos con características comunes y suelen estructurarse en jerarquías mediante la relación *rdfs:subClassOf*. Las instancias son elementos concretos de las clases y se relacionan con ellas mediante la propiedad *rdf:type*. Finalmente, las propiedades son predicados RDF que definen relaciones entre instancias o entre instancias y clases, y suelen caracterizarse por su dominio y rango. Las propiedades pueden estructurarse de forma jerárquica mediante la relación *rdfs:subPropertyOf*.

SPARQL

SPARQL (acrónimo recursivo de *SPARQL Protocol and RDF Query Language*) [122] es un lenguaje de consultas sobre grafos RDF que permite recuperar tripletes o subgrafos RDF que cumplen ciertos patrones.

OWL

OWL (*Web Ontology Language*) [59] permite representar de manera explícita el significado de los términos que componen un vocabulario, así como las relaciones entre dichos términos. Esta representación de términos y sus relaciones se denomina comúnmente ontología.

OWL se basa en RDF y RDFS pero proporciona muchos más recursos para describir ontologías: descripciones de clases usando combinaciones lógicas de otras clases (intersección, unión, complemento), restricciones de cardinalidad sobre las propiedades, propiedades algebraicas (reflexividad, transitividad, ...), etc.

La especificación original de OWL define tres variantes del lenguaje: OWL Lite, OWL DL y OWL Full. Cada uno de ellas es una extensión de la anterior en el que se aumenta la expresividad del lenguaje. Por tanto, las ontologías OWL Lite son ontologías válidas OWL DL y estas, a su veces, son ontologías válidas OWL Full. Desafortunadamente, el aumento de expresividad del lenguaje conlleva una aumento en el coste computacional necesario en los procesos de razonamiento, por lo que cada aplicación debe elegir la variante de OWL que mejor se adapta a sus necesidades:

- **OWL Lite** permite definir jerarquías usando restricciones sencillas. La expresividad del lenguaje se mantiene muy limitada para favorecer

la gestión eficiente de las bases de conocimiento. OWL Lite puede ser una buena opción para representar tesauros y taxonomías.

- **OWL DL** está pensado para los usuarios que quieren la mayor expresividad posible y a la vez necesitan que los procesos de razonamiento sean *decidibles*. OWL DL incluye todas las constructoras de OWL pero establece ciertas limitaciones en su uso (por ejemplo, una clase no puede ser instancia de otra clase). OWL DL se basa en una familia de lenguajes formales, las lógicas descriptivas [6], con una semántica bien definida y propiedades formales bien estudiadas.
- **OWL Full** es la variante más expresiva de OWL pero no proporciona ningún tipo de garantías computacionales. En OWL Full es posible definir entidades que sean a la vez clases, propiedades e instancias. La consecuencia es que los razonadores que usan OWL Full tienden a ser correctos pero incompletos.

Resumen de la pila de lenguajes

La gran cantidad de lenguajes y tecnologías que se usan en la web semántica (aquí sólo hemos nombrado algunas) puede hacer difícil comprender lo que aporta cada una de ellas al conjunto. A modo de resumen:

- XML permite representar y compartir información estructurada a nivel sintáctico, sin imponer restricciones sobre el significado de los documentos.
- XML Schema describe la estructura de otros documentos XML y extiende XML con tipos de datos.
- RDF representa modelos de datos en forma de grafo dirigido donde se establecen relaciones entre entidades.
- RDFS permite definir clases de entidades y propiedades entre ellas. Además proporciona una semántica sencilla que permite definir jerarquías de clases y propiedades.
- OWL incorpora nuevas constructoras del lenguaje para definir ontologías como relaciones lógicas entre clases, restricciones de cardinalidad, etc.

2.1.3. Lógicas descriptivas

Las lógicas descriptivas [6] (DLs, del inglés *Description Logics*) son *subconjuntos decidibles de la lógica de predicados*. Permiten representar el conocimiento terminológico de un dominio de forma estructurada usando jerarquías conceptuales. Las DLs surgen como evolución natural de los sistemas de

marcos y redes semánticas [23] que, a diferencia de las DLs, no contaban con una semántica formal bien definida. Durante los últimos años las DLs han adquirido gran popularidad, jugando un papel esencial en la formalización de ontologías para la web semántica.

En los siguientes apartados profundizamos en esta familia de lenguajes formales, especialmente en la variante que corresponde al lenguaje OWL-DL de la web semántica. Comenzaremos resumiendo la notación que se utiliza para distinguir las distintas DLs. Más tarde resumiremos la sintaxis y semántica concretas del lenguaje formal equivalente a OWL-DL. A continuación describiremos los servicios de razonamiento básicos que se suelen realizar con este tipo de bases de conocimiento. Finalmente, describiremos un lenguaje formal de consultas que nos permitirá recuperar información.

Antes de continuar debemos advertir al lector que las siguiente secciones están escritas a modo de resumen y sólo tienen como objetivo introducir la sintaxis que usaremos más adelante en la memoria. Si no está familiarizado con las lógicas descriptivas le recomendamos que consulte antes otros textos de carácter más pedagógico [64, 23, 6].

2.1.3.1. Expresividad

El término lógicas descriptivas da nombre a toda una familia de lenguajes, cada uno de ellos con una expresividad y complejidad diferentes en función del tipo de constructoras disponibles [1]. Las distintas variantes se identifican mediante un código de letras que representan los operadores disponibles (ver tabla 2.1).

OWL-DL es una variante sintáctica de la lógica $\mathcal{SHOIN}^{(\mathcal{D})}$, mientras que OWL-Lite corresponde a $\mathcal{SHIF}^{(\mathcal{D})}$. En las siguientes secciones nos centraremos en $\mathcal{SHOIN}$, es decir, el equivalente a OWL-DL pero sin *datatypes* (números, cadenas, fechas, ...).

2.1.3.2. Sintaxis

Sean N_C, N_R y N_I conjuntos no vacíos y disjuntos de *conceptos atómicos*, *roles atómicos* e *individuos* respectivamente.

El conjunto de roles de $\mathcal{SHOIN}$ es $N_R \cup \{R^- | R \in N_R\}$ donde R^- denota la *inversa del rol* atómico R. Sean $R, S \in N_R$ roles atómicos. Un *axioma de inclusión de roles* es una expresión de la forma $R \sqsubseteq S$, y un *axioma de transitividad* es una expresión de la forma $Trans(R)$.

Sea $A \in N_C$ un concepto atómico, C y D dos conceptos cualesquiera, R un rol, n un número natural y $a \in N_I$ un individuo. Los conceptos de $\mathcal{SHOIN}$ se define de manera inductiva usando las constructoras mostradas en la tabla 2.2.

[1] Consultar `http://www.cs.manchester.ac.uk/~ezolin/dl/` para un resumen actualizado de las distintas complejidades

$\mathcal{AL}$	Atomic negation, concepto intersection, universal restrictions, limited existencial quantification
$\mathcal{F}$	Functional properties
$\mathcal{E}$	Full existential qualification
$\mathcal{U}$	Concept union
$\mathcal{C}$	Complex concept negation
$\mathcal{S}$	An abbreviation for $\mathcal{ALC}$ with transitive roles.
$\mathcal{H}$	Role hierarchy
$\mathcal{R}$	Limited complex role inclusion axioms, reflexivity and irreflexivity, role disjointness
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse properties
$\mathcal{N}$	Cardinality restrictions
$\mathcal{Q}$	Qualified cardinality restrictions
$^{(\mathcal{D})}$	Use of datatype properties, data values or data types.

Tabla 2.1: Códigos más comunes para identificar las distintas DLs.

Conceptos	Sintaxis
Concepto atómico	A
Intersección	$C \sqcap D$
Unión	$C \sqcup D$
Complemento	$\neg C$
Restricción existencial	$\exists R.C$
Restricción de valor	$\forall R.C$
Restricción at-most	$\leq nR$
Restricción at-least	$\geq nR$
Nominales	$\{a\}$

Tabla 2.2: Conceptos de la lógica $\mathcal{SHOIN}$.

Un *axioma de inclusión de conceptos* es una expresión de la forma $C \sqsubseteq D$. Una *TBox* $\mathcal{T}$ es un conjunto finito de axiomas de inclusión de conceptos, axiomas de inclusión de roles y axiomas de transitividad.

Un *axioma de equivalencia de conceptos* tiene la forma $C \equiv D$ y es simplemente una abreviatura de $C \sqsubseteq D$ y $D \sqsubseteq C$. Un axioma $C \sqsubseteq D$ se denomina *definición primitiva* si C es un nombre de concepto y se denomina *concepto general de inclusión* (CGI) si es un concepto complejo. Decimos que una *TBox* es *acíclica* si la definición de ninguno de sus conceptos depende directa o indirectamente de si mismo.

Una *ABox* $\mathcal{A}$ es un conjunto finito de axiomas de la forma $C(a)$, $R(a, b)$, $\neg R(a, b)$, $a = b$ y $a \neq b$, siendo C un concepto, R un rol, y a y b individuos.

Una *base de conocimiento* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ es un vector donde $\mathcal{T}$ es una TBox y $\mathcal{A}$ es una ABox. En ocasiones usaremos las siglas KB como abreviatura de base de conocimiento (en inglés *Knowledge Base*).

Finalmente, usaremos la notación $\mathcal{K} \cup \{\alpha\}$ para representar la adición

$(\neg C)^{\mathcal{I}}$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
$(C \sqcap D)^{\mathcal{I}}$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$(C \sqcup D)^{\mathcal{I}}$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\{a\}$	$\{a^{\mathcal{I}}\}$
$(\exists R.C)^{\mathcal{I}}$	$\{x \mid \exists y : (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
$(\forall R.C)^{\mathcal{I}}$	$\{x \mid \forall y : (x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
$(\geq nR)^{\mathcal{I}}$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}}\} \geq n\}$
$(\leq nR)^{\mathcal{I}}$	$\{x \mid \sharp\{y \mid (x,y) \in R^{\mathcal{I}}\} \leq n\}$
$(Inv(R))^{\mathcal{I}}$	$\{(x,y) \mid (y,x) \in R^{\mathcal{I}}\}$

Tabla 2.3: Semántica de conceptos y roles

Sintaxis	Semántica
$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
$Tans(R)$	$(R^{\mathcal{I}})^{+} \subseteq R^{\mathcal{I}}$
$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
$R(a,b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
$\neg R(a,b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin R^{\mathcal{I}}$
$a = b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
$a \neq b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

Tabla 2.4: Semántica de los axiomas

del axioma α a la base de conocimiento. Dependiendo del tipo de axioma se añadirá a la TBox o la ABox.

2.1.3.3. Semántica

Una interpretación es un vector $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ donde $\Delta^{\mathcal{I}}$ es un conjunto no vacío llamado dominio de la interpretación, y $\cdot^{\mathcal{I}}$ es la función de interpretación. La función de interpretación mapea cada concepto atómico $C \in N_C$ a un subconjunto de $\Delta^{\mathcal{I}}$, cada rol atómico $R \in N_R$ a un subconjunto de $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, y cada individuo $a \in N_I$ a un elemento de $\Delta^{\mathcal{I}}$.

La función de interpretación se extiende a los conceptos y roles generales tal y como se detalla en la tabla 2.3.

Una interpretación $\mathcal{I}$ satisface un axioma α ($\mathcal{I} \models \alpha$) si se cumple la condición correspondiente de la tabla 2.4. Decimos que una interpretación es un modelo de una TBox $\mathcal{T}$ (resp. ABox $\mathcal{A}$) si satisface todos los axiomas de $\mathcal{T}$ (resp. $\mathcal{A}$). Finalmente, $\mathcal{I}$ es un modelo de una base de conocimiento $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, denotado como $\mathcal{I} \models \mathcal{K}$ si $\mathcal{I}$ es modelo de $\mathcal{T}$ y $\mathcal{A}$.

Las bases de conocimiento se interpretan usando la hipótesis de mundo abierto (OWA, del inglés *Open World Assumption*), es decir, no podemos asumir nada sobre la información que no aparece en la KB de manera explícita o implícita. Una KB representa una *descripción parcial de un estado real desconocido*, o lo que es lo mismo, puede admitir varios modelos (o in-

cluso infinitos modelos) que se corresponden con todos aquellos estados que concuerdan con la descripción parcial. Las bases de datos, por el contrario, utilizan la hipótesis de mundo cerrado (CWA, del inglés *Closed World Assumption*) en la que cualquier sentencia que no se sabe cierta se asume que es falsa.

2.1.3.4. Problemas de inferencia

El problema de inferencia básico de las DLs es comprobar la *consistencia* de una base de conocimiento. Decimos que una base de conocimiento $\mathcal{K}$ es consistente si tiene algún modelo. Otros problemas de inferencia son:

- *Satisfactibilidad de conceptos*: un concepto C es satisfactible respecto a $\mathcal{K}$ si existe un modelo $\mathcal{I}$ de $\mathcal{K}$ tal que $C^{\mathcal{I}} \neq \emptyset$.
- *Subsunción de conceptos*: un concepto C es subsumido por otro concepto D respecto a $\mathcal{K}$ si para cada modelo $\mathcal{I}$ de $\mathcal{K}$ se cumple $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.
- *Equivalencia de conceptos*: dos conceptos C y D son equivalentes respecto a $\mathcal{K}$ si C subsume a D y D subsume a C.
- *Clasificación de una base de conocimiento*: calcular todas las relaciones de subsunción entre conceptos atómicos para crear una jerarquía conceptual.
- *Reconocimiento de instancias*: un individuo a es una instancia del concepto C respecto a $\mathcal{K}$ si para todo modelo $\mathcal{I}$ de $\mathcal{K}$ se cumple $a^{\mathcal{I}} \in C^{\mathcal{I}}$.
- *Realización*: encontrar los conceptos atómicos más específicos de los que cada individuo es instancia.

2.1.3.5. Consultas conjuntivas

Una *consulta conjuntiva DL*, $Q(\mathbf{x}, \mathbf{y})$, es una conjunción de términos de la forma $C(x)$, $R(x, y)$ y $\neg R(x, y)$, donde C es un concepto, R un role, y x e y son nombres de individuos de N_I, elementos del dominio o nombres de variables sacados de los conjuntos $\mathbf{x}$ e $\mathbf{y}$. Los conjuntos $\mathbf{x}$ e $\mathbf{y}$ contienen, respectivamente, las variables *distinguibles* y *no distinguibles* de la consulta. Una *consulta booleana* es una consulta sin variables distinguibles. Usaremos $Q(\mathbf{y})$ para denotar las consultas booleanas, y $Q()$ para denotar una consulta booleana sin variables.

Una *sustitución de variables* es una función que relaciona nombres de variables con nombres de individuos y nombres de variables. Si α es un término de una consulta y σ una sustitución, escribiremos $\alpha[\sigma]$ para denotar el término que resulta de sustituir las variables de α de acuerdo con σ. Si

$Q(\mathbf{x}, \mathbf{y})$ es una consulta entonces $Q(\mathbf{x}[\sigma], \mathbf{y})$ representa la consulta que resulta de sustituir las variables distinguibles de Q de acuerdo con σ.

Sea $VI(Q)$ el conjunto de variables e individuos de Q. Una interpretación $\mathcal{I}$ es un *modelo de una consulta booleana* $Q(\mathbf{y})$, escrito $\mathcal{I} \models \exists \mathbf{y} : Q(\mathbf{y})$ o abreviado $\mathcal{I} \models Q$, si existe una sustitución $\sigma : VI(Q) \rightarrow \Delta^{\mathcal{I}}$ tal que $\sigma(a) = a^{\mathcal{I}}$ para cada individuo $a \in VI(Q)$, y $\mathcal{I} \models \alpha[\sigma]$ para cada átomo α de la consulta. La consulta Q es una *consecuencia lógica* de la base de conocimiento $\mathcal{K}$, escrito $\mathcal{K} \models Q$, si todo modelo de $\mathcal{K}$ satisface Q.

Ahora pasamos al caso de consultas con variables distinguibles. Una *respuesta* a una consulta $Q(\mathbf{x}, \mathbf{y})$ con respecto a una base de conocimiento $\mathcal{K}$ es una sustitución σ que relaciona las variables distinguibles de la consulta $\mathbf{x}$ con individuos de $\mathcal{K}$ de manera que la consulta $Q(\mathbf{x}[\sigma], \mathbf{y})$ es consecuencia lógica de $\mathcal{K}$. Mientras que para interpretar consultas booleanas permitimos que la sustitución relacione variables con elementos arbitrarios del dominio, para responder consultas con variables distinguibles exigimos que las variables distinguibles se relacionen con individuos de la base de conocimiento.

El *conjunto de respuestas* (*answer set*) de una consulta Q con respecto a $\mathcal{K}$, denotado $Q(\mathcal{K})$, es el conjunto que contiene todas las respuestas de Q con respecto a $\mathcal{K}$. Como las consultas booleanas no tienen variables distinguibles, el conjunto de respuestas para una consulta booleana será bien vacío, indicando que la consulta no es consecuencia lógica de la KB, o el conjunto que contiene la sustitución vacía, indicando que la consulta sí es consecuencia lógica de la KB.

Una consulta $Q_1(\mathbf{x}, \mathbf{y_1})$ es *subsumida por* (*contenida en*) otra consulta $Q_2(\mathbf{x}, \mathbf{y_2})$ con respecto a una TBox $\mathcal{T}$ denotado $Q_1 \sqsubseteq_{\mathcal{T}} Q_2$, si para cada posible ABox $\mathcal{A}$ y base de conocimiento $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ se cumple que $Q_1(\mathcal{K}) \subseteq Q_2(\mathcal{K})$. En nuestra definición de subsunción de consultas suponemos que Q_1 y Q_2 comparten el mismo conjunto de variables distinguibles.

2.1.3.6. Ontologías OWL-DL y bases de conocimiento DL

El término ontología se suele reservar para descripciones semánticas de un dominio consensuadas, de manera independiente al lenguaje en el que se describan. En este trabajo, sin embargo, cuando usamos el término ontología nos referimos a ontologías OWL-DL.

Además, usamos los términos ontología OWL-DL y base de conocimiento DL de manera indistinta. Debe entenderse en estos casos, que cualquier ontología OWL-DL puede verse como una base de conocimiento semántico descrito en el lenguaje formal DL equivalente.

Por último, la terminología que se utiliza en OWL para nombrar las distintas entidades de una ontología es distinta que la que se usa en las DLs para nombrar las entidades equivalentes. Nosotros usaremos los términos concepto - clase y rol - propiedad como sinónimos a lo largo de la memoria.

2.1.4. Herramientas de gestión de ontologías

Hasta ahora hemos visto que las ontologías de la web semántica cuentan con un lenguaje estándar de representación y están fundamentadas sobre una base formal sólida. Sin embargo, para hacer accesible y práctica está tecnología no basta con eso, también es necesario disponer de un conjunto de herramientas que faciliten los procesos de creación y mantenimiento de las ontologías.

Cuando se construyen grandes bases de conocimiento surgen importantes problemas relacionados con la representación, validación y evolución de dicho conocimiento:

- *Navegación.* Para gestionar grandes bases de conocimiento es útil disponer de herramientas que nos muestren de manera intuitiva el conocimiento. Resulta útil poder visualizar la información con distintos niveles de detalle, así como poder elegir y aislar la parte de la base de conocimiento relevante en cada caso.

- *Extracción/Integración.* Uno de los objetivos de las ontologías es potenciar la reutilización del conocimiento. Resulta imprescindible disponer de herramientas que faciliten la combinación y extracción del conocimiento disponible en las ontologías existentes.

- *Validación.* Al representar grandes cantidades de información es frecuente cometer errores. Es muy útil disponer de herramientas que automáticamente detecten inconsistencias en la base de conocimiento o conceptos que no se pueden satisfacer.

- *Diseño colaborativo.* Las bases de conocimiento pueden ser grandes y complejas, por lo que resulta útil disponer de herramientas de tipo colaborativo que permitan a varios usuarios editar simultáneamente la información.

- *Evolución.* Las bases de conocimiento evolucionan a lo largo del tiempo, por lo que resulta útil contar con herramientas de gestión de versiones.

Existen multitud de herramientas relacionadas con la gestión de ontologías. En [73] se enumeran muchos de estos sistemas clasificados por su arquitectura, sistemas de representación de conocimiento, servicios de inferencia, y distintas características de usabilidad (representaciones gráficas, colaboración, librerías de ontologías, ...).

Nosotros nos vamos a centrar en dos herramientas que describimos en las siguientes secciones: *Protégé*, un intuitivo y potente editor de ontologías, y *Pellet*, un potente razonador de DLs.

2.1.4.1. Editores de ontologías

The Protégé Ontology Editor and Knowledge Acquisition System [99], en adelante *Protégé*, es una plataforma gratuita y de código abierto orientada a la construcción de ontologías y aplicaciones basadas en conocimiento. Ha sido desarrollada por el Centro de Investigación Bio-médica de la Escuela de Medicina de la Universidad de Standford, y hoy en día cuenta con el respaldo de una gran comunidad que incluye desarrolladores, investigadores, agencias gubernamentales y empresas, además de más de 126.000 usuarios registrados.

Protégé esta desarrollado en Java usando una arquitectura extensible basada en *plug-ins*, que permite el desarrollo rápido de prototipos y aplicaciones. El núcleo de Protégé define las estructuras necesarias para representar, visualizar y manipular ontologías, mientras que los *plug-ins* se utilizan para extender la funcionalidad del entorno adaptándolo a las necesidades específicas de cada dominio.

La plataforma Protégé permite dos formas distintas de modelar ontologías:

- *Protégé-Frames* [44] es un editor que permite la creación de ontologías usando una aproximación basada en *frames*, usando el protocolo *Open Knowledge Base Connectivity* (OKBC). En este modelo, una ontología consiste en un conjunto de clases organizadas jerárquicamente, un conjunto de slots asociados a las clases para describir sus propiedades y sus relaciones, y un conjunto de instancias o individuos que pertenecen a uno o más conceptos y establecen los valores concretos de las propiedades.

- *Protégé-OWL* [76] es un editor que permite construir ontologías para la web semántica usando el lenguaje OWL. Existen diversos razonadores que se integran en este entorno usando la arquitectura de *plug-ins* de Protégé. Usando estos sistemas, y gracias a la semántica formal de OWL, podemos derivar hechos que no están explícitamente representados en la ontología pero que son consecuencia lógica del conocimiento representado. Los razonadores también nos permiten detectar inconsistencias en la base de conocimiento, facilitando el desarrollo de grandes ontologías.

2.1.4.2. Razonamiento y validación

Existen muchos razonadores diferentes tanto a nivel académico como comercial: FaCT++ [140], KAON2 [67], RacerPro [52], CEL [7], Hermit [94], etc. Aunque todos ellos ofrecen los servicios básicos de razonamiento que describimos en el apartado 2.1.3.4, cada uno de ellos tiene características avanzadas especiales y está optimizado para un sublenguaje DL concreto.

Nosotros nos vamos a centrar en Pellet [126], un razonador de código abierto implementado en Java que permite razonar de manera incremental cuando se realizan pequeños cambios en la ABox [56]. Este razonador es capaz de gestionar ontologías OWL 2 [142] e incorpora varias técnicas de optimización que mejoran su eficiencia a la hora de gestionar nominales [124] y resolver consultas conjuntivas [125].

Las características más interesantes de este razonador son:

- *Varias interfaces.* Podemos utilizar los servicios de razonamiento desde la línea de comandos, a través de las bibliotecas de programación OWL-API [66] y Jena [1], usarlo integrado como plug-in en el editor de ontologías Protégé, o usarlo como servidor DIG [53].

- *Optimizado para consultas conjuntivas* (SPARQL-DL). Pellet contiene múltiples optimizaciones para responder consultas conjuntivas. Además puede responder consultas SPARQL de tipo SELECT, CONSTRUCT y ASK por sí mismo, y consultas de tipo DESCRIBE o que usen OPTIONAL y FILTER si se usa como razonador integrado en la arquitectura de Jena.

- *Razonamiento con datatypes.* Este razonador permite razonar con todos los datatypes predefinidos en XML Schema (numéricos, cadenas, fecha y hora, ...) y con algunos tipos definidos por el usuario como extensiones de los anteriores.

- *Gestión de reglas SWRL.* Pellet puede gestionar ontologías extendidas mediante reglas representadas en el lenguaje SWRL.

- *Análisis de ontologías.* Pellet incorpora heurísticas que permiten detectar ontologías marcadas como OWL Full que pueden ser expresadas usando OWL-DL. En algunos casos permite corregir estas ontologías automáticamente.

- *Depuración de ontologías.* La detección de conceptos que no se pueden satisfacer es un servicio de razonamiento básico que proporcionan todos los razonadores. Sin embargo, pocos razonadores ayudan al usuario en el diagnóstico de las causas de la inconsistencia así como su resolución. Pellet es capaz de calcular los axiomas y hechos de la ontología responsables de la inconsistencia, y cómo las dependencias entre conceptos hacen que el error se propague.

- *Razonamiento incremental.* Este razonador gestiona eficientemente bases de conocimiento dinámicas mediante el uso de algoritmos incrementales de clasificación y chequeo de consistencia.

2.2. Planificación

Esta segunda parte del capítulo la dedicaremos a los problemas de planificación. Comenzaremos con una pequeña introducción en la que describiremos el modelo *clásico* de planificación y las simplificaciones en las que se basa. Después repasaremos tres de la aproximaciones más conocidas: búsqueda en el espacio de estados, planificación jerárquica y planificación basada en casos. A continuación repasaremos el lenguaje estándar de planificación y veremos los cambios que ha sufrido a lo largo de los años. Finalmente, recopilamos los trabajos más significativos relativos al uso de ontologías para modelar conocimiento de planificación, tratando de conectar los dos grandes temas de este capítulo.

2.2.1. Introducción

La Planificación es un importante área de la Inteligencia Artificial que estudia la parte del razonamiento asociada al acto [100]. Un planificador es un sistema que elige y organiza acciones, intentando anticipar sus consecuencias, de manera que al ejecutarlas en cierto orden se alcancen unos objetivos preestablecidos de la mejor manera posible.

Nosotros estamos interesados en planificación independiente del dominio, es decir, sistemas que reciben como entrada la descripción del dominio y del problema concreto dentro de ese dominio que se desea resolver. En estos sistemas, los dominios y problemas se describen usando lenguajes formales de propósito general. A las descripciones simbólicas de los dominios y problemas se los denomina *modelo del dominio* y *modelo del problema*. La expresividad de los lenguajes empleados y su complejidad determinan los problemas que se pueden describir y la dificultad de razonar sobre dichas descripciones.

Un modelo conceptual sencillo (pero incompleto) que nos puede ayudar a visualizar un problema de planificación es el modelo de sistema de transición de estados. Un sistema de transición de estados es un vector $\Sigma = (S, A, E, \gamma)$ donde:

- $S = \{s_1, s_2, \ldots\}$ es un conjunto finito o recursivamente enumerable de estados.
- $A = \{a_1, a_2, \ldots\}$ es un conjunto finito o recursivamente enumerable de acciones.
- $E = \{e_1, e_2, \ldots\}$ es un conjunto finito o recursivamente enumerable de eventos.
- $\gamma : S \times A \times E \to 2^S$ es la función de transición de estados.

Los sistema de transición de estados tratan de modelar los cambios que se producen en un sistema cuando un agente realiza una acción o cuando se

produce un evento. Los eventos no los produce ningún agente, representan cambios producidos por dinámicas internas al sistema. Podemos representar un sistema de transición de estados mediante un grafo dirigido cuyos nodos se corresponden con los estados de S. La aristas del grafo responden a posibles transiciones entre estados debidas a acciones y/o eventos. Las aristas del grafo vienen determinadas por la función de transición de estados, en concreto existe una arista etiquetada como (a_i, e_j) desde el nodo s_u al nodo s_v sii $\gamma(s_u, a_i, e_j) = s_v$.

Para poder abordar los problemas de planificación de manera efectiva es necesario realizar ciertas simplificaciones. En concreto, en lo que se conoce como planificación *clásica* se asume un modelo restringido basado en las siguientes hipótesis:

1. Σ *finito*. El conjunto de posibles estados S es finito.

2. Σ *completamente observable*. En todo momento se conoce el estado actual del sistema.

3. Σ *determinista*. Si una acción es aplicable en un cierto estado, el sistema evoluciona a otro estado de manera unívoca; y lo mismo pasa con los eventos ($|\gamma(s, a, e)| \leq 1$).

4. Σ *estático*. El conjunto de posibles eventos E es vacío, es decir, las únicas transiciones entre estados son producidas como consecuencia de acciones.

5. *Objetivos restringidos*. El objetivo de un problema de planificación consiste en alcanzar alguno de los estados definidos en un conjunto finito de estados objetivo S_g. No se contempla la posibilidad de imponer restricciones adicionales sobre las trayectorias o sobre la estructura de los planes.

6. *Planes secuenciales*. Un plan solución es una secuencia finita y linealmente ordenada de acciones.

7. *Tiempo implícito*. Las acciones y los eventos no tienen duración, produciendo transiciones entre estados instantáneas.

8. *Planificación off-line*. No ocurre ningún cambio en Σ mientras el planificador está buscando un plan solución.

Resolver un problema de planificación en este modelo restringido equivale a resolver un problema de búsqueda de caminos en un grafo dirigido, un problema bien estudiado y para el que existen algoritmos eficientes. Sin embargo, incluso en dominios sencillos el grafo Σ resultante es tan grande que no es posible representarlo de manera explícita. Surge, por tanto, la

necesidad de buscar otras representaciones que nos permitan describir subconjuntos interesantes de S de manera compacta, y la necesidad de definir heurísticas que guíen el proceso de búsqueda.

Este modelo restringido se puede relajar permitiendo describir problemas más interesantes desde el punto de vista práctico, pero también más difíciles de resolver. Modificaciones comunes a este modelo son, por ejemplo, la inclusión de objetivos complejos (funciones de coste a minimizar, estados que se deben evitar, restricciones sobre la estructura de los planes, ...), acciones con duración (concurrencia), estructuras de planes más flexibles (ordenes parciales entre las acciones), etc. También es posible incorporar fuentes de incertidumbre en distintas partes del modelo: estados parcialmente observables, acciones no deterministas, sistemas dinámicos (con eventos), incorporación de probabilidades al modelo, etc.

2.2.2. Técnicas de planificación

Existen muchas técnicas de planificación distintas en función del tipo de conocimiento disponible, la forma en que se representa el espacio de búsqueda y la manera de recorrerlo. Ghallab, Nau y Traverso han creado un magnifico texto orientado a la enseñanza [100] que describe las principales técnicas relacionadas con la planificación automática. A continuación enumeramos algunas de ellas junto con el capítulo de referencia del libro anterior donde se desarrollan.

Las aproximaciones más clásicas al problema de planificación lo modelan como una búsqueda en el espacio de estados (capítulo 4) o en el espacio de planes (capítulo 5). Otras técnicas más modernas se valen de grafos para definir mejores heurísticas de búsqueda (capítulo 6) o re-codifican el problema para utilizar técnicas de satisfactibilidad proposicional (capítulo 7). Otras aproximaciones incorporan los conceptos de tiempo (capítulo 13 y 14) o modelos probabilísticos (capítulo 16).

Nosotros nos vamos a centrar en tres de la aproximaciones muy conocidas a las que haremos referencia a lo largo de este trabajo: búsqueda en el espacio de estados, planificación HTN y planificación basada en casos.

2.2.2.1. Búsqueda en el espacio de estados

La planificación como búsqueda en el espacio de estados es la aproximación más sencilla e intuitiva. El espacio de búsqueda corresponde con el grafo dirigido que usamos para representar el sistema de transición de estados. Cada nodo representa un posible estado del mundo y cada arco se corresponde con una transición entre estados producida como consecuencia de ejecutar una acción. Un plan representa un camino en dicho grafo que lleva desde el estado inicial a un estado objetivo.

A pesar de ser una representación sencilla, durante mucho tiempo no se

descubrieron buenas heurísticas que guiaran el proceso de búsqueda. Como consecuencia, otras aproximaciones, como la búsqueda en el espacio de planes, tomaron protagonismo. Sin embargo, el panorama ha cambiado durante los últimos años y algunos de los planificadores más rápidos actuales usan esta aproximación [63, 88, 20, 10, 61].

2.2.2.2. Planificación HTN

En planificación HTN (del inglés *Hierarchical Task Network*) el objetivo no es alcanzar un estado que cumpla ciertas condiciones sino resolver un conjunto de *tareas*. La descripción de un dominio HTN incluye dos mecanismos de resolución de tareas: *métodos* y *operadores*. Los métodos resuelven tareas complejas (*no primitivas*) descomponiéndolas en subtareas más sencillas, y los operadores resuelven las tareas sencillas (*primitivas*) y modifican el estado actual del sistema. El proceso de planificación consiste, por tanto, en descomponer las tareas objetivo en subtareas más sencillas de manera recursiva mediante métodos, hasta alcanzar tareas primitivas que pueden resolverse usando operadores.

Las ideas básicas de la planificación HTN provienen de los trabajos de Sacerdoti [121] y Tate [137] en los años 70. Los primeros pasos hacia un modelo teórico fueron dados por Yang [147] y Kambhampati y Hendler [72], y un modelo completo fue desarrollado por Erol et al. [37].

La planificación HTN es más expresiva que la planificación clásica [38], y los planificadores HTN han demostrado ser muy rápidos si se codifica correctamente el conocimiento. El principal problema de este enfoque consiste, precisamente, en la necesidad de tener conocimiento adicional del dominio que permita describirlo de manera jerárquica, una información que en ocasiones puede no estar disponible.

Este tipo de planificación ha sido de los más usados en aplicaciones reales [144], y se ha aplicado en dominios tan diversos como la gestión de situaciones de crisis [18], planes de evacuación [96], juego del bridge [128], robótica [93], videojuegos [97], etc. Algunos de los planificadores HTN más conocidos son: Nonlin [137], SIPE-2 [145], O-Plan [30, 138], UMCP [37] y SHOP2 [101].

2.2.2.3. Planificación basada en casos

En planificación basada en casos (CBP, del inglés *Case-Based Planning*) los problemas se intentan resolver adaptando las soluciones que se aplicaron a problemas similares en el pasado. La idea que subyace a esta aproximación es la hipótesis de que problemas similares suelen tener soluciones similares. Existen dos razones básicas para usar CBP: dificultad para modelar completamente el dominio y/o eficiencia.

Los planificadores generativos necesitan descripciones completas del dominio en el que trabajan con una semántica clara. La obtención y codifica-

ción del conocimiento específico de cada dominio es una de las tareas más complicadas a la hora de crear aplicaciones prácticas. Además, los expertos suelen encontrar más sencillo describir ejemplos de problemas y soluciones, que explicar modelos globales aplicables a cualquier situación. En este tipo de dominios complejos las aproximaciones basadas en casos cobran especial importancia. La base de casos es una recopilación de experiencias y estas experiencias suelen contener, de manera implícita, conocimiento sobre el dominio que no somos capaces de modelar de manera explícita.

Con respecto a la eficiencia, debemos tener en cuenta que la planificación es un proceso computacionalmente costoso incluso en dominios sencillos. Si un planificador tiene que resolver con frecuencia un determinado tipo de problemas, puede ser mucho más rápido recuperar y adaptar soluciones aplicadas a problemas similares que intentar crear la solución desde cero.

Algunos de los planificadores basados en casos más conocidos son: PRIAR [72], PRODIGY/Analogy [141], SPA [58], DerSNLP [68], PARIS [14], CaPER [75], CAPlan/CbC [98], DIAL [78], CHEF [57] y HICAP [95].

El ciclo básico de un sistema CBR consiste en cuatro fases [2]: recuperar uno o varios casos relacionados con el problema actual, adaptar sus soluciones al nuevo contexto, revisar si la solución propuesta es válida, e incorporar la experiencia como un nuevo caso en la base de casos.

La fase de recuperación tiene como cometido recuperar casos de la base de casos que contengan problemas similares al actual. Para ello se utiliza una medida de similitud entre problemas, asumiendo que cuanto más se parezcan los problemas recuperados al problema actual más fácil será adaptar sus soluciones. Existen muchas medidas de similitud propuestas en la literatura. Las medidas de similitud estáticas sólo tienen en cuenta los problemas que se comparan a la hora de calcular su similitud; las dinámicas, en cambio, tienen en cuenta también los episodios problema-solución anteriores. Ejemplos de sistemas con medidas de similitud estáticas son PRIAR, PRODIGY/Analogy, SPA, DerSNLP, PARIS. Ejemplos de sistemas con medidas de similitud dinámicas son CaPER, CAPlan/CbC.

Las técnicas de adaptación de casos se pueden clasificar en dos grandes grupos: *analogía por transformación* (*transformational analogy*) [28] y *analogía por derivación* (*derivational analogy*) [29]. En las adaptaciones basadas en analogía por transformación, las soluciones de los casos recuperados se modifican tratando de adaptarlas al nuevo problema. Las adaptaciones pueden consistir en eliminar o re-ordenar las acciones del plan, cambiar el valor de sus parámetros, etc. En las técnicas de adaptación basadas en analogía por derivación, sin embargo, los casos no contienen directamente la solución de los problemas sino las trazas de las decisiones que se tomaron para llegar a obtener la solución. La adaptación, por tanto, consiste en volver a tomar las mismas decisiones en el contexto del nuevo problema. Los planificadores PRIAR, SPA y DIAL utilizan adaptación basada en analogía por transforma-

ción, mientras que los planificadores PRODIGY/Analogy, PARIS, DerSNLP y CAPlan/CbC usan analogía por derivación.

Algunos planificadores basados en casos no tienen componente generativa (DIAL, CHEF) por lo que necesitan una gran base de casos para comportarse bien. Otros planificadores sí incorporan una componente generativa (PRODIGY/Analogy, DerSNLP, PARIS, DerUCP) y pueden beneficiarse de ambas técnicas, pero para ello necesitan una descripción completa del dominio. También existen planificadores basados en casos que suplen la carencia de conocimiento del dominio mediante interacción con un experto humano (HICAP, CAPlan/CbC).

En cuanto al rendimiento, Nebel and Koehler [103] demostraron que bajo ciertas hipótesis de adaptación conservativa, la adaptación de planes puede ser exponencialmente más costosa que la generación de planes desde cero. Este resultado parecía contradecir algunos resultados empíricos en los que los planificadores basados en casos resultaban más rápidos que los generativos en ciertos dominios. Posteriormente Au et al. [5] demostraron que: (1) la analogía por derivación no cumple los supuestos establecidos en el trabajo de Nebel y Koehler, y (2) que, además, nunca incrementa el espacio de búsqueda y en algunos casos puede disminuirlo exponencialmente.

2.2.3. PDDL

El lenguaje PDDL (*Planning Domain Definition Language*) se creó en 1998 [46] para poder comparar los distintos planificadores que participaban en la primera competición internacional de planificación. La versión original del lenguaje resultó demasiado pretenciosa y se redujo considerablemente de cara a la siguiente competición dos años más tarde [9], acercando PDDL a lo que los planificadores de la época eran capaces de resolver. A partir de entonces, el lenguaje se ha ido enriqueciendo poco a poco en cada competición. A continuación resumimos los cambios más importantes:

- PDDL 2.1 [42] (2002) incorpora nuevas características como son el tratamiento de *fluents* numéricos y la representación explícita de tiempo en las acciones.

- PDDL 2.2 [36] (2004) añade los conceptos de predicados derivados y *timed initial literals*. Los predicados derivados son predicados cuya verdad depende de la de otros predicados básicos. Los *timed initial literals* son hechos que se harán ciertos en un determinado instante de tiempo, independientemente de las acciones que el planificador escoja.

- PDDL 3.0 [45] (2006) añade restricciones y preferencias expresadas en una lógica temporal restringida. Las restricciones permiten imponer ciertas condiciones sobre las estructura de los planes válidos, mientras

que las preferencias establecen objetivos deseados pero no imprescindibles, introduciendo métricas que hacen algunos planes válidos mejores que otros.

- PDDL 3.1 [62] (2008) añade el concepto de *object fluents* de manera análoga a los *fluents* numéricos de PDDL 2.1.

Hoy en día PDDL es el estándar de facto para representar dominios de planificación, aunque también ha sido criticado por no cumplir las características ideales desde el punto de vista de la ingeniería del conocimiento [87].

Los dominios PDDL contienen, esencialmente, la descripción del vocabulario del dominio (predicados, funciones, constantes) y los operadores de planificación. Los operadores de planificación, a su vez, se describen mediante precondiciones y efectos. Las precondiciones establecen las condiciones necesarias para poder aplicar la acción correspondiente, y los efectos describen los cambios que se producirán en el estado del sistema tras ejecutar la acción. Finalmente, los dominios PDDL contienen una descripción de las características que un planificador debe tener para poder resolver problemas en este dominio. Usando este mecanismo es fácil extender el lenguaje con nuevas características sin romper la compatibilidad hacia atrás.

Los problemas PDDL contienen una referencia al dominio que da contexto al problema, la descripción del estado inicial y los objetivos del problema. Los objetivos se suelen describir mediante las condiciones que un estado debe cumplir para ser considerado un estado objetivo, pero también es posible definir objetivos más complejos imponiendo restricciones sobre la estructura de los planes, funciones de coste a minimizar, o incluso objetivos deseables pero no imprescindibles.

La figura 2.2 muestra un ejemplo sencillo. En este dominio tenemos un robot que puede moverse entre habitaciones y que, además, tiene brazos con los que puede coger pelotas. Los problemas que definiremos en este dominio consisten en encontrar la secuencia de acciones que debe efectuar el robot para trasladar pelotas entre habitaciones.

La definición del dominio PDDL contiene la descripción de los predicados del dominio y la definición de las tres operaciones que puede efectuar el robot: moverse de un cuarto a otro (*move*), coger una pelota con un brazo (*pick*) y soltar la pelota (*drop*). Cada uno de estos operadores tiene sus precondiciones y efectos. Por ejemplo, para poder mover al robot desde *?from* hasta *?to* se deben cumplir ciertas condiciones: *?from* y *?to* deben ser habitaciones y el robot debe estar en la primera. Tras ejecutar la acción correspondiente (una vez los parámetros se han vinculado a objetos concretos del estado), el robot dejará de estar en la primera habitación para encontrarse en la segunda.

La definición del problema PDDL comienza definiendo los símbolos de constante que servirán para definir el estado inicial, en el que hay dos habi-

```
(define (domain gripper−strips)
   (:requirements :strips)
   (:predicates (room ?r) (ball ?b) (gripper ?g) (at−robby ?r)
                (at ?b ?r) (free ?g) (carry ?o ?g))
   (:action move
      :parameters (?from ?to)
      :precondition (and (room ?from) (room ?to) (at−robby ?from))
      :effect (and (at−robby ?to) (not (at−robby ?from))))
   (:action pick
      :parameters (?obj ?room ?gripper)
      :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
                         (at ?obj ?room) (at−robby ?room) (free ?gripper))
      :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
               (not (free ?gripper))))
   (:action drop
      :parameters (?obj ?room ?gripper)
      :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
                         (carry ?obj ?gripper) (at−robby ?room))
      :effect (and (at ?obj ?room) (free ?gripper)
               (not (carry ?obj ?gripper)))))

(define (problem strips−gripper2)
   (:domain gripper−strips)
   (:objects rooma roomb ball1 ball2 left right)
   (:init (room rooma) (room roomb)
          (ball ball1) (ball ball2)
          (gripper left) (gripper right)
          (at−robby rooma)
          (free left) (free right)
          (at ball1 rooma) (at ball2 rooma))
   (:goal (at ball1 roomb)))
```

Figura 2.2: Ejemplo de problema PDDL (Manuela Veloso)

taciones (*rooma* y *roomb*), cada una con una pelota (*ball1* y *ball2*), el robot tiene dos brazos libres (*left* y *right*), y el robot está en la primera habitación. El objetivo consiste en llevar la pelota *ball1* desde la habitación *rooma* hasta *roomb*.

El lenguaje PDDL se diseñó pensando en problemas deterministas, pero también existen extensiones para modelar problemas con incertidumbre. En la última competición de planificación se definieron 4 familias de problemas en función de la fuente de incertidumbre [26]: *Fully Observable Probabilistic* (FOP), *Non Observable Non-Deterministic* (NOND), *Fully Observable Non-Deterministic* (FOND) y *Partially-Observable Probabilistic Planning* (POP).

2.2.4. Planificación y ontologías

Algunas condiciones necesarias para extender el uso de la planificación en entornos reales son: (1) aumentar la expresividad de los lenguajes de modelado para poder resolver problemas en dominios más complejos; (2) facilitar la adquisición, validación y reutilización del conocimiento; y (3) facilitar la integración de los planificadores dentro de infraestructuras de negocio complejas, resolviendo los problemas de interacción con otros sistemas inteligentes.

El uso de ontologías para representar dominios de planificación puede aportar algunas soluciones a estos problemas, pero también plantea nuevos retos que deben ser estudiados con detenimiento. En primer lugar, las ontologías representan conocimiento de carácter estático mientras que los lenguajes de planificación como PDDL centran su atención en la acciones y los cambios que estas producen. Por otra parte, las ontologías representan el conocimiento usando lenguajes muy expresivos e interpretan dicho conocimiento bajo la hipótesis de mundo abierto, es decir, suponiendo que el conocimiento representado es incompleto. En planificación, por el contrario, los lenguajes de modelado suelen estar mucho más limitados y trabajan bajo la cómoda hipótesis de mundo cerrado.

En general hay dos grandes líneas en la investigación relacionadas con el uso de ontologías para representar conocimiento de planificación, dependiendo de si las ontologías se usan o no *durante* el proceso de planificación. En la primera alternativa, las ontologías se usan para modelar una gran base de conocimiento con toda la información del dominio, y a partir de ahí se extrae la información que cada subsistema necesita para operar. Si queremos utilizar planificación necesitamos, por tanto, extraer el conocimiento que el planificador va a necesitar y traducirlo al lenguaje adecuado. El proceso de traducción suele requerir la intervención de un experto y conlleva una perdida de información debido a la diferencia de expresividad de los lenguajes. A cambio se consiguen dominios más sencillos donde poder aplicar técnicas de planificación eficientes. La segunda alternativa consiste en utilizar un planificador que sea capaz de utilizar directamente el conocimiento ontológico y aproveche toda esa información a la hora de resolver problemas.

En la conferencia internacional de planificación del año 2005 (ICAPS05) tuvo lugar un taller dedicado a estudiar el papel de las ontologías para planificación. La mayoría de los trabajos que se presentaron en el taller usaban ontologías para representar conocimiento general del dominio que luego se traducía de forma parcial para utilizar planificadores estándar. A continuación describimos algunos de estos trabajos.

En [86] se estudia cómo aliviar el proceso de ingeniería de conocimiento necesario para modelar un dominio de planificación reutilizando modelos de dominios que ya están representados como ontologías OWL. La idea es aprovechar todo el esfuerzo que ya se ha invertido para modelar esos dominios. La traducción se realiza mediante heurísticas, con lo que se obtiene un dominio

de planificación incompleto que debe ser validado y refinado por un experto.

El trabajo descrito en [90] trata sobre los conflictos que surgen debido a la gran expresividad de los lenguajes de representación de ontologías, frente a la relativa pobreza exigida por los planificadores para resolver problemas de manera eficiente. En el trabajo se propone un arquitectura en la que el conocimiento del dominio y del estado del mundo se representa usando un lenguaje expresivo (KIF), que se traduce de manera parcial a un modelo menos expresivo (PDDL) cuando un agente necesita resolver un determinado problema usando planificación. El coste de esta traducción es que el planificador no será capaz de hallar todas las soluciones al problema original, pero a cambio, puede hallar algunas soluciones de manera eficiente.

Por último, [4] describe un sistema que ayuda a los turistas a planificar sus rutas en una ciudad. Para mejorar su usabilidad, se trata de un sistema *on-line* al que se puede acceder mediante dispositivos móviles. En este caso los autores parten de una base de conocimiento formalizada en CLIPS que describe la información de interés turístico de la ciudad y hacen una traducción *ad-hoc* de este dominio concreto a PDDL.

La segunda alternativa que hemos comentado consiste en aprovechar las características de expresividad y razonamiento de los lenguajes de representación de ontologías *durante* el proceso de planificación. Usando lógicas descriptivas podemos sacar partido de las definiciones lógicas de las clases para inferir automáticamente nuevas relaciones clase-subclase y clasificar los individuos como instancias de las clases adecuadas. De esta manera podemos aprovechar las descripciones de acciones, planes y objetivos en dominios ricos en conocimiento, tanto durante la generación de planes como en el reconocimiento y evaluación de los mismos.

Esta idea se desarrolla en el trabajo de Gil [47] en el que se estudian tres sistemas que usan DLs para representar conocimiento de planificación. En concreto se identifican 4 posibles tipos de ontologías:

1. *Taxonomías de objetos.* Permiten razonar sobre el estado del planificador, que se describe enumerando los tipos de los objetos y sus relaciones.

2. *Taxonomías de acciones.* De este modo se puede razonar sobre tipos de acciones a distintos niveles de abstracción.

3. *Taxonomías de planes.* Permiten razonar sobre la subsunción de planes parcialmente ordenados.

4. *Taxonomías de objetivos.* Para poder razonar usando representaciones expresivas de los objetivos y sus parámetros.

Más adelante, en el capítulo 4, detallaremos algunas de las ideas que encontramos en los trabajos más recientes en este área. En ellos se utilizan

ontologías para representar el estado del planificador (taxonomías de objetos), y se estudian los problemas relativos a actualizar ABoxes [82] así como la complejidad computacional de varios problemas relacionados con planificación utilizando distintos tipos de DLs [8, 92, 91]. Por otra parte, en el capítulo 6 describiremos como usar DLs para razonador sobre los objetivos de los planes (taxonomías de objetivos) y de ese modo identificar cuándo podemos reutilizar planes en una aproximación a la planificación basada en casos.

En el área de la composición de servicios web semánticos, el trabajo de Sirin [123] define un algoritmo de planificación HTN-DL que combina las ideas de la planificación HTN y la representación del conocimiento usando DLs. En este caso las ontologías se utilizan para tres cosas: representar el estado del mundo y comprobar los servicios web que se pueden invocar; representar taxonomías de tareas; e identificar los servicios web que son capaces de resolver cada tarea mediante subsunción de precondiciones y postcondiciones. En este trabajo se distinguen, además, dos tipos de servicios web: los puramente informativos, que sólo sirven para conseguir información sobre el estado actual del mundo (pero no lo modifican), y los que sí tienen efectos que alteran el estado y por tanto deben ser invocados con precaución.

Otro trabajo relacionado que presenta un compromiso de expresividad es [22], donde los autores presentan un planificador que admite conocimiento expresado mediante ontologías OWL. Para evitar invocar al razonador durante el proceso de planificación, trabajan con un subconjunto de OWL que tiene complejidad polinomial y puede evaluarse mediante reglas. En este caso se usa un razonador DL para generar el conjunto de reglas *off-line*, y de esta manera evitan costosos procesos de inferencia durante la resolución de problemas.

Para concluir, nos gustaría resaltar el esfuerzo que está realizando la comunidad de DLs para optimizar la gestión de bases de conocimiento dinámicas [21, 55] y el interés creciente en la integración de DLs y formalismos de acciones [47, 8, 92, 91], lo que nos lleva a pensar que en el futuro próximo veremos grandes avances en este campo.

2.3. Conclusiones

Las ontologías son un artefacto muy interesante desde el punto de vista de la representación de conocimiento, ya que permiten modelar dominios estructurados de manera intuitiva y extensible. Además, promueven que los modelos de los dominios se compartan y reutilicen en distintos sistemas, abaratando los costes de creación de sistemas inteligentes.

En la primera parte de este capítulo, hemos repasado las tecnologías y herramientas desarrolladas en el ámbito de la web semántica para facilitar la creación y gestión de ontologías. Comenzamos describiendo los distintos

lenguajes estándar para representar ontologías y nos centramos en OWL-DL, un lenguaje que ofrece un compromiso interesante entre expresividad y complejidad de razonamiento. Después, estudiamos los lenguajes formales conocidos como lógicas descriptivas, que son la base formal en la que se apoya OWL-DL. Finalmente, enumeramos algunas de las herramientas que nos ayudan a gestionar este tipo de conocimiento: editores visuales y razonadores.

La segunda parte del capítulo la hemos dedicado a la planificación. Comenzamos repasando tres aproximaciones muy conocidas al problema: búsqueda en el espacio de estados, planificación HTN y planificación basada en casos. Más tarde presentamos el lenguaje estándar de planificación, PDDL, y mostramos un ejemplo concreto en un dominio muy sencillo. Por último hemos recopilado los trabajos significativos relativos al uso de ontologías para modelar conocimiento de planificación, clasificándolos en dos grandes grupos: los que extraen el conocimiento de ontologías y lo traducen al lenguaje del planificador *off-line*, y los que son capaces de aprovechar la información ontológica durante el proceso de planificación.

En el siguiente capítulo veremos cómo podemos utilizar ontologías para modelar dominios de planificación con un vocabulario complejo y mucho conocimiento asociado, y los problemas que surgen cuando intentamos traducir todo ese conocimiento a un lenguaje de planificación como PDDL.

Capítulo 3

Dominios de planificación ricos en conocimiento

Cuando intentamos utilizar técnicas de planificación para resolver problemas del mundo real nos encontrarnos de manera natural con dominios ricos en conocimiento. La resolución de estos problemas puede implicar la gestión de grandes bases de conocimiento en las que se combina información que proviene de diversas fuentes. Para poder afrontar este tipo de retos la comunidad de planificación debe trabajar en dos frentes complementarios: (1) mejorar los procesos de adquisición y gestión del conocimiento asociado a los dominios de planificación, y (2) buscar mecanismos que permitan integrar los actuales algoritmos de búsqueda y representaciones de conocimiento específico del dominio representado en lenguajes expresivos.

En este capítulo nos centramos en la parte relativa a la adquisición de conocimiento, y proponemos el uso de ontologías OWL-DL para modelar dominios de planificación con un vocabulario complejo y mucho conocimiento asociado. En concreto utilizaremos ontologías para representar el conocimiento de carácter estático: el vocabulario del dominio y los tipos y relaciones de las entidades. Esta aproximación tiene dos ventajas fundamentales: aprovechamos la expresividad de las DLs para modelar dominios complejos, y las herramientas de gestión de ontologías (editores, razonadores, repositorios, ...) de la web semántica. La parte de planificación propiamente dicha, es decir, la definición de problemas y el recorrido del espacio de búsqueda, la explicaremos en el capítulo siguiente.

Comenzaremos este capítulo recuperando una interesante definición de ingeniería del conocimiento relativa al área de planificación, y veremos cómo el uso de ontologías puede aliviar algunos de los procesos involucrados en la adquisición y gestión del conocimiento. También analizaremos los dominios que se usaron en la última competición internacional de planificación, concluyendo que son dominios con estructuras muy sencillas y poco conocimiento asociado. Por último, mostraremos un dominio con un vocabulario mucho

más rico pero que resulta muy sencillo de modelar usando ontologías.

Más tarde estudiaremos los problemas que surgen cuando intentamos traducir el conocimiento representado en una ontología OWL-DL al lenguaje de planificación PDDL. También propondremos un esquema de traducción parcial que puede resultar útil en ocasiones. Al traducir el conocimiento perdemos parte de la información original debido a la diferencia de expresividad de ambos lenguajes. Para concluir el capítulo, enumeraremos algunas de las ventajas que tendría un sistema de planificación que utilice DLs para representar el conocimiento.

3.1. Ontologías para modelar conocimiento de planificación

En 2003, la *European Network of Excellence in AI Planning* definió la Ingeniería del Conocimiento en Planificación ([17], capítulo 7) como el proceso que comprende:

1. la adquisición, validación, verificación y mantenimiento de los modelos de dominios de planificación;

2. y la selección de los mecanismos de planificación adecuados y su integración con el modelo del dominio para construir una aplicación.

En el citado documento se reconoce que el área de planificación tiene características únicas que la diferencian de otras áreas en las que también se estudian sistemas basados en conocimiento. A grandes rasgos las principales diferencias son: (1) que el conocimiento fundamental en planificación orbita alrededor de un conjunto de acciones que modifican objetos; y (2) que el objetivo final de un modelo del dominio es formar parte de un sistema que construye planes, en contraste con las típicas tareas de clasificación o diagnóstico comunes en otros sistemas basados en conocimiento. A pesar de estar diferencias, concluye, *la comunidad de planificación debe estudiar las técnicas y herramientas relacionadas con la gestión de conocimiento desarrolladas en otras áreas*, y estudiar cómo adaptarlas e integrarlas en los sistemas de planificación.

Precisamente nuestra propuesta consiste en tratar de aprovechar las tecnologías y herramientas desarrolladas en el ámbito de la web semántica para representar dominios de planificación ricos en conocimiento. Nosotros pensamos que el uso de ontologías OWL-DL aporta ciertas ventajas:

- Estas ontologías se basan en lenguajes formales expresivos, y permiten definir vocabularios mucho más complejos de los que estamos acostumbrados a encontrar en los dominios de planificación.

- El conocimiento en OWL-DL se interpreta usando la hipótesis de mundo abierto, lo que permite razonar con conocimiento que se sabe *a priori* incompleto. Si más tarde añadimos nuevo conocimiento a la ontología, tenemos garantizado que no se invalidará ninguno de los razonamientos hechos hasta ese momento.

- Las ontologías pueden utilizarse para definir un vocabulario común con el que homogeneizar el tratamiento de la información que proviene de distintas fuentes, y de esta manera facilitar la comunicación entre sistemas.

- El auge de las ontologías en la web semántica ha propiciado la creación de toda una infraestructura orientada a facilitar la adquisición y gestión del conocimiento:

  - Los *lenguajes estándar* y *repositorios de ontologías* facilitan el modelado de dominios, y permiten aprovechar el esfuerzo que otros han invertido para representar entornos similares.
  - Los *editores de ontologías visuales* proporcionan entornos amigables que simplifican el acceso a estas tecnologías. Además, permiten gestionar grandes proyectos que, debido a su complejidad, serían inviables sin estas herramientas.
  - Los *razonadores* permiten *detectar y corregir inconsistencias* en las bases de conocimiento. Además, son sistemas muy optimizados capaces de *gestionar grandes cantidades de información de manera eficiente*.

Sin embargo, el uso de ontologías en planificación también plantea nuevos problemas que deben ser estudiados con detenimiento. Las ontologías se suelen utilizar para representar conocimiento de carácter estático, mientras que los lenguajes de planificación como PDDL centran su atención en la acciones y los cambios que éstas producen. Además, las ontologías representan el conocimiento usando lenguajes muy expresivos e interpretan dicho conocimiento bajo la hipótesis de mundo abierto, es decir, suponiendo que el conocimiento representado es incompleto. En planificación, sin embargo, los lenguajes de modelado suelen estar mucho más limitados y el conocimiento se interpreta, por lo general, usando la cómoda hipótesis de mundo cerrado.

En las siguientes secciones profundizaremos en estos dos problemas. Primero repasaremos la información que aparece en los dominios y problemas de planificación, tratando de determinar qué parte podemos representar usando ontologías. A continuación veremos un ejemplo concreto de ontología que podemos utilizar como base para definir un dominio de planificación con un vocabulario interesante. Finalmente, estudiaremos las dificultades que surgen al intentar traducir el conocimiento representado en la ontología a PDDL.

	Estático	Dinámico
Dominio	Tipos Constantes dominio Funciones Predicados	Operadores Restricciones dominio Preferencias dominio
Problema	Constantes problema Estado inicial	Restricciones problema Preferencias problema Objetivos

Tabla 3.1: Conocimiento de carácter estático y dinámico.

3.1.1. Conocimiento estático y dinámico

En planificación independiente del dominio siempre se distingue entre conocimiento del dominio y conocimiento del problema. El conocimiento del dominio comprende la información común a cualquier problema, es decir, define un marco común y delimita el tipo de problemas que se puede definir. El conocimiento del problema completa la información del dominio definiendo una situación específica en la que se persigue alcanzar ciertos objetivos concretos.

En PDDL (descrito en la sección 2.2.3) el modelo del dominio contiene esencialmente la descripción del vocabulario del dominio (tipos, constantes, funciones y predicados) y los operadores de planificación. El modelo del problema completa esta información con constantes adicionales, la descripción del estado inicial y los objetivos que se persiguen. Tanto la descripción del dominio como la del problema pueden contener, además, restricciones y preferencias que delimitan o priorizan los posibles planes solución.

Nosotros vamos a plantear una clasificación adicional del conocimiento transversal a la anterior, distinguiendo entre conocimiento de carácter estático y conocimiento de carácter dinámico (ver tabla 3.1).

Entendemos por conocimiento de carácter estático todo el conocimiento necesario para describir el estado del sistema en un determinado instante de tiempo. Dentro de esta clasificación incluimos el vocabulario del dominio (tipos, constates, predicados), posibles axiomas que relacionan y definen los términos del vocabulario, y los literales que describen el estado del sistema (por ejemplo el estado inicial).

El conocimiento de tipo dinámico, por el contrario, está relacionado con la evolución del sistema a lo largo del tiempo. Este tipo de conocimiento comprende la descripción de los operadores de planificación, los objetivos del problema, posibles restricciones estructurales sobre los planes y las preferencias a la hora de recorrer el espacio de búsqueda.

Históricamente, la comunidad de planificación ha centrado sus esfuerzos en la gestión del conocimiento dinámico. Es razonable, al fin y al cabo la parte de conocimiento más importante desde el punto de vista de la planificación se

Dominio	Tipos	Ctes	Pred.	Funciones	Acciones
elevators-strips	5	0	8	3	6
openstacks-adl	3	0	7	1	4
openstacks-strips	3	10 - 200	8	1	12 - 202
parcprinter-strips	7	26 - 45	11	1	23 - 36
pegsol-strips	1	0	5	1	3
scanalyzer-strips	2	0	6	1	4
sokoban-strips	5	0	6	1	3
transport-strips	6	0	5	2	3
woodworking-strips	17	11	15	5	13

Tabla 3.2: Dominios de planificación IPC 2008 (*Sequential Satisficing track*).

encuentra en los operadores y cómo combinarlos para hallar planes solución. Como consecuencia, disponemos de algoritmos de búsqueda muy eficientes que sólo funcionan en dominios con vocabularios sencillos. Por ejemplo, es común asumir que los predicados lógicos son totalmente independientes, lo que por una parte limita el conocimiento que se puede representar pero por otra facilita enormemente la gestión de las bases de conocimiento.

La misma evolución de PDDL muestra la preferencia de la comunidad hacia el conocimiento de carácter dinámico. Las distintas revisiones del lenguaje han ido incorporando nueva información relacionada con las acciones (duración, efectos condicionales, indeterminismo, ...) y con los planes (restricciones estructurales, funciones de coste y preferencias, ...). La parte de conocimiento estático, sin embargo, se sigue manteniendo lo más sencilla posible: predicados lógicos independientes, y representación del estado mediante un conjunto de átomos interpretados usando mundo cerrado. De hecho, la única aportación importante durante los últimos años ha sido la incorporación de los predicados derivados o axiomas PDDL [139] que, aunque son importantes desde el punto de la eficiencia, no aportan nada a la expresividad del lenguaje.

Si analizamos el vocabulario de los dominios que se usaron en la última competición internacional de planificación (*IPC 2008 - Sequential Satisficing track*) podemos ver que son dominios modelados con una estructura y vocabulario sencillos. La tabla 3.2 muestra el número de tipos, constantes, predicados, funciones y operadores definidos en cada dominio (cuando hay varias versiones de un dominio, la tabla muestra el número mínimo y máximo de elementos definidos). Cabe destacar que en ninguno de ellos se utilizan predicados derivados, es decir, existe total independencia entre los predicados. [1]

Sin embargo, el hecho de utilizar vocabularios sencillos no debe llevarnos a pensar que los problemas de planificación definidos en estos dominios son

[1] La tabla no incluye el dominio *cybersec-strips* porque su descripción no usa variables (es *ground strips*) y por tanto consideramos que no es comparable con el resto.

sencillos de resolver. Este tipo de problemas basa su dificultad en la longitud de la solución, es decir, en la explosión combinatoria del espacio de búsqueda inherente a este tipo de problemas (ver [100] capítulo pag. 59 para un resumen de la complejidad en problemas de planificación clásica). Nosotros, por el contrario, estamos interesados en problemas de planificación en los que la dificultad depende más de la gestión del conocimiento asociado al dominio que del recorrido inteligente del espacio de búsqueda.

3.1.2. Ontologías para modelar conocimiento estático

La comunidad de la web semántica ha dedicado mucho esfuerzo durante los últimos años para facilitar la adquisición y gestión de grandes bases de conocimiento semántico. Todo este esfuerzo y dinero invertido ha permitido el desarrollo de modelos teóricos sobre los que posteriormente se han creado lenguajes estándar y herramientas que facilitan los procesos de adquisición y gestión de conocimiento.

Uno de los dominios más conocidos modelados usando OWL-DL es una ontología sobre pizzas que se utiliza en un tutorial de la Universidad de Manchester [64]. Este didáctico tutorial describe cómo podemos usar el editor de ontologías Protégé (figura 3.1) y las distintas constructoras de OWL-DL para definir de manera natural las entidades del dominio, en este caso *ingredientes*, *bases* y *tipos de pizza*. Además, durante la creación de la ontología se explica cómo podemos usar cualquiera de los razonadores integrados en la herramienta para detectar y corregir errores en la base de conocimiento. Resulta interesante comprobar que, siguiendo este tutorial, una persona sin conocimientos de lógicas descriptivas puede aprender a utilizar OWL-DL y modelar un dominio de cierta complejidad en muy poco tiempo. La ontología final cuenta con 100 clases, 8 propiedades, 5 individuos y más de 600 axiomas sólo para caracterizar las clases.

Intentar modelar un dominio como el de la pizzas sin utilizar Protégé nos habría llevado mucho más tiempo y el proceso habría sido propenso a errores. Este entorno de edición de ontologías ofrece varias herramientas que facilitan enormemente el proceso. En primer lugar, trae varios razonadores DL integrados que permiten verificar la consistencia de la base de conocimiento. Además, permite visualizar de manera gráfica distintas partes de la ontología usando tanto el modelo asertado como el inferido. Estas representaciones visuales de la ontología resultan muy útiles para comprender y revisar la estructura del dominio. También permite realizar consultas sobre la base de conocimiento y recuperar la información que necesitemos. Finalmente, existen multitud de complementos que se pueden instalar en el entorno usando la arquitectura de *plug-ins* de Protégé.

Es interesante observar que desde el punto de vista de los razonadores OWL-DL esta ontología es una base de conocimiento sencilla. Al fin y al cabo este dominio se utiliza como ejemplo para enseñar a modelar conocimiento

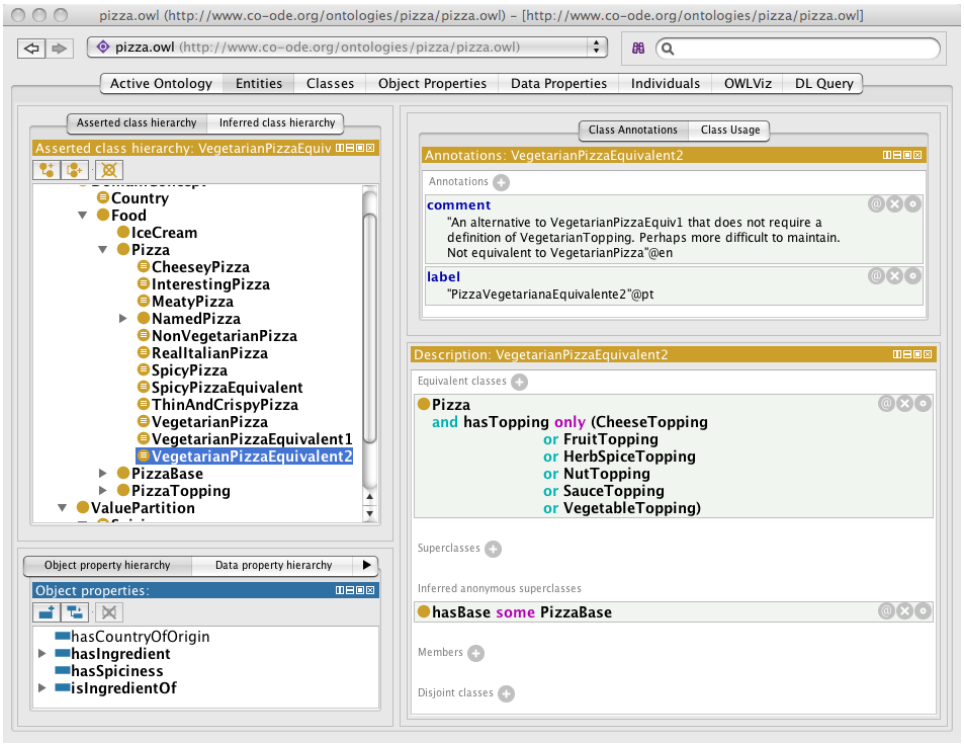


Figura 3.1: Editor de ontologías Protégé.

usando OWL-DL. Sin embargo, el vocabulario que define esta ontología es muy complejo si lo comparamos con el que suelen tener los dominios de planificación (ver tabla 3.2), no sólo por el número de predicados implicados sino por la cantidad de axiomas que los relacionan.

3.1.3. Ontología para una pizzería

Usando las facilidades que ofrece OWL-DL y las herramientas tipo Protégé es sencillo definir nuevos dominios reutilizando el conocimiento formalizado previamente por otras personas y que se comparte mediante repositorios de ontologías. A modo de ejemplo, nosotros hemos extendido la ontología de las pizzas con nuevos conceptos (bebidas, menús, ...) para modelar una pizzería. La tabla 3.3 muestra el número de entidades y axiomas que contiene la ontología final.

En el siguiente capítulo veremos cómo podemos utilizar esta ontología como base para modelar un dominio de planificación donde los problemas consisten en combinar distintos ingredientes para construir pizzas y menús con ciertas características. La parte terminológica de la ontología o TBox sirve para definir el vocabulario del dominio de planificación, mientras la

Entities	
Classes	138
Object properties	7
Data properties	0
Individuals	0
Total	145
DL expressivity	SHIN

Class axioms	
SubClass	143
Equivalent	49
Disjoint classes	444
Total	636

Object property axioms	
Sub object properties	4
Functional object properties	3
Inverse func. object properties	5
Transitive object properties	1
Object property domain	4
Object property range	5
Total	22

Anotations	
Entity anotations	112

Tabla 3.3: Entidades y axiomas de la ontología de una pizzería.

parte asertiva o ABox sirve para modelar situaciones concretas. Es importante entender la estructura de este dominio, porque a lo largo de la memoria propondremos varios ejemplos basados en él. A continuación repasamos las partes principales de la ontología y mostramos algunos de los conceptos y axiomas que contiene [2].

Las figuras 3.2 y 3.3 muestran parte del vocabulario disponible para describir distintos tipos de ingredientes y pizzas. Los ingredientes se clasifican en función de su tipo (*Meat*, *Fish*, *Cheese*, *Fruit*, ...) y lo picantes que son (*Mild*, *Medium*, *Hot*). Las pizzas, a su vez, se clasifican en función del número de ingredientes que contienen (*PizzaWith0Toppings*, *PizzaWith1Topping*, ...), el tipo de ingredientes (*MeatyPizza*, *VegetarianPizza*, *SpicyPizza*, ...), el tipo de base que utilizan (*ThinBase*, *DeepAndPanBase*) y su tamaño (*Small*, *Regular*, *Familiar*). Además, la ontología contiene la definición de varias pizzas estándar que podemos encontrar en cualquier restaurante: *Margherita*, *FourSeasons*, *Napoletana*, etc.

Estas jerarquías visuales se construyen a partir de los axiomas de la TBox. Los conceptos *primitivos* de la ontología se describen usando condiciones necesarias ($\sqsubseteq$), y los conceptos *definidos* mediante condiciones necesarias y suficientes ($\equiv$). A continuación mostramos algunos de estos axiomas a modo de ejemplo.

$$Spiciness \equiv Hot \sqcup Medium \sqcup Soft \tag{3.1}$$
$$SpicyTopping \equiv Topping \sqcap Hot \tag{3.2}$$
$$Olives \sqsubseteq Vegetable \sqcap Soft \tag{3.3}$$
$$Mozzarella \sqsubseteq Cheese \sqcap Soft \tag{3.4}$$
$$Vegetarian \equiv Cheese \sqcup Fruit \sqcup HerbSpice \sqcup \tag{3.5}$$

[2]La ontología completa está accesible en la dirección `http://gaia.fdi.ucm.es/people/antonio/data/tesis/pizzeria.owl`

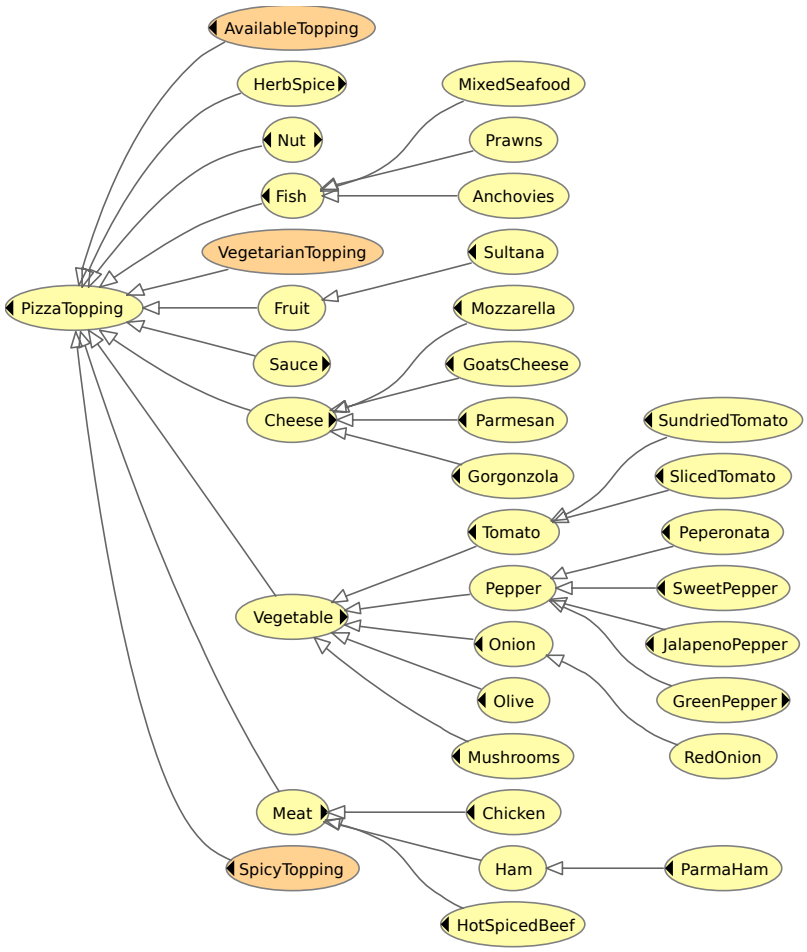


Figura 3.2: Jerarquía parcial de ingredientes.

$$
\begin{aligned}
 & & Nut \sqcup Sauce \sqcup Vegetable & \\
SpicyPizza & \equiv & Pizza \sqcap \exists hasTopping.SpicyTopping & (3.6)\\
MeatyPizza & \equiv & Pizza \sqcap \exists hasTopping.Meat & (3.7)\\
VegetarianPizza & \equiv & Pizza \sqcap \forall hasTopping.VegetarianTopping & (3.8)\\
InterestingPizza & \equiv & Pizza \sqcap (\geq 3 hasTopping) & (3.9)\\
Marguerita & \equiv & Pizza \sqcap (= 2 hasTopping) \sqcap & (3.10)\\
 & & \exists hasTopping.Cheese \sqcap \exists hasTopping.Tomato &
\end{aligned}
$$

El primer axioma define 3 grados de picante distintos que nos servirán para clasificar a los alimentos según su sabor. El axioma (3.2) define el concepto ingrediente picante como la intersección de los dos conceptos implicados. Los axiomas (3.3) y (3.4) describen dos tipos de ingredientes concretos indicando su tipo padre en la ontología y lo picantes que son. El axioma 3.5 define

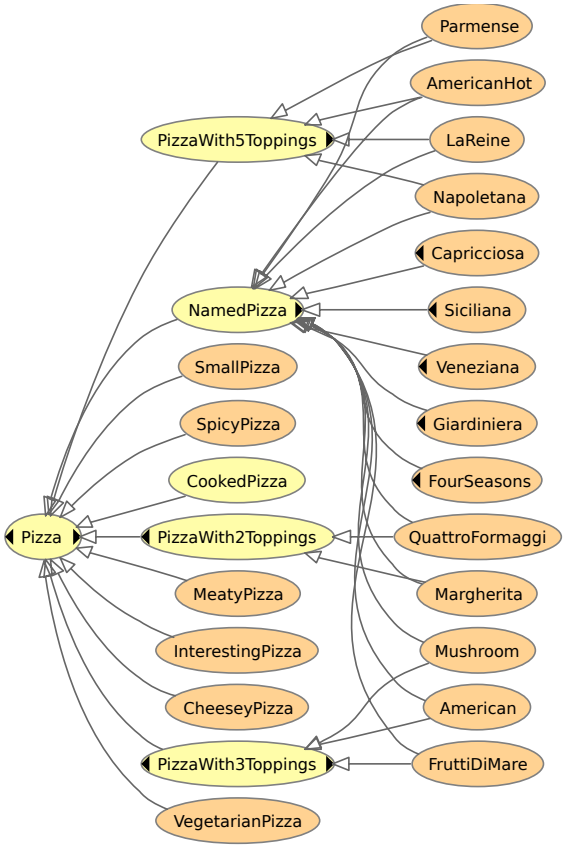


Figura 3.3: Algunas de las pizzas definidas en la ontología.

cuando considereamos que un ingrediente es vegetariano. El axioma (3.6) define que una pizza se considera picante si contiene al menos un ingrediente picante. De manera similar, el axioma (3.7) define el concepto de pizza con carne. El axioma (3.8) indica que una pizza es vegetariana si sólo contiene ingredientes vegetarianos. El axioma (3.9) nos dice que una pizza es interesante si tiene al menos 3 ingredientes. Finalmente, el axioma (3.10) define que las pizzas margaritas tienen dos ingredientes de tipo tomate y queso.

Otra característica interesante de las ontologías es que la jerarquía conceptual nos permite describir situaciones usando distinto nivel de detalle. Por ejemplo, los conceptos *Mozzarella* y *Marguerita* representan un tipo concreto de ingrediente y un tipo concreto de pizza, mientras que *SpicyTopping* y *VegetarianPizza* son conceptos abstractos que permiten describir situaciones más generales donde no se conocen o no importan los ingredientes específicos. Esta riqueza de vocabulario nos permite definir escenarios usando distintos niveles de detalle, por ejemplo:

$$SpicyPizza(p), MeatyPizza(p), hasTopping(p, t1), Olives(t1),$$
$$hasTopping(t2), Mozzarella(t2)$$

Es importante resaltar que debido a la hipótesis de mundo abierto la situación anterior se interpreta como una descripción parcial e incompleta, y por tanto sólo indica que existe una pizza picante con carne que tiene aceitunas y queso mozzarella. No se asume nada más, es decir, la pizza puede tener cualquier tipo de base o incluso otros ingredientes distintos de *t1* y *t2*. De hecho, dada la información anterior un razonador inferirá que debe existir al menos otro ingrediente en la pizza ya que p es una pizza picante con carne y ni las aceitunas ni la mozzarena son picantes ni tipos de carne. De esta forma, la pizza p debe tener 3 o más ingredientes y será clasificada como una instancia del concepto *InterestingPizza*. Este es sólo un pequeño ejemplo del tipo de inferencias que un razonador de DLs puede calcular a partir de una base de conocimiento aparentemente simple.

Por último, vamos a mostrar cómo recuperar información mediante consultas DL. Debemos tener en cuenta que, si usamos un razonador, la información recuperada contendrá tanto los datos asertados inicialmente en la ontología como los inferidos a partir de los axiomas. Por ejemplo, podemos recuperar todas las pizzas con al menos un ingrediente de tipo vegetariano usando la siguiente consulta:

$$Pizza(?x), hasTopping(?x, ?y), Vegetarian(?y)$$

3.2. Traducción de ontologías OWL-DL a PDDL

Representar conocimiento estático usando ontologías tiene ventajas evidentes desde el punto de vista de la ingeniería del conocimiento. Desafortunadamente, los mecanismos que proporciona PDDL para representar este tipo de conocimiento son demasiado básicos, y cuando intentamos traducir una ontología OWL-DL a PDDL no podemos mantener la semántica original. Por lo general, la versión PDDL del vocabulario del dominio sólo contendrá una pequeña parte del conocimiento original. Pero no sólo es una traducción parcial, durante el proceso se añadirá, de manera implícita, conocimiento que no estaba en la ontología original como consecuencia de la distinta intepretación del conocimiento que hacen ambos lenguajes: mundo abierto en OWL-DL y mundo cerrado en PDDL.

La consecuencia es que, aunque intentemos semi-automatizar la traducción, al final siempre necesitamos de un experto del dominio que conozca ambos lenguajes y dedique el tiempo necesario para seleccionar y representar de manera adecuada el conocimiento que el planificador necesita. Este

proceso, además de ser complejo y laborioso para ontologías de cierto tamaño, sufre los problemas relativos al mantenimiento de dos representaciones del dominio distintas que deben estar siempre sincronizadas.

Web-PDDL [89] es un intento de ampliar el lenguaje PDDL para poder representar ontologías. Usando la misma sintaxis de PDDL, se amplia el lenguaje permitiendo representar jerarquías conceptuales complejas (donde cada tipo puede tener varios padres) y axiomas del dominio de manera análoga a como se hace en OWL. Aunque este tipo de propuestas es muy interesante, además de relevante para nuestro trabajo, por el momento no parece haber tenido demasiada repercusión en la comunidad de planificación.

A continuación escribiremos las principales diferencias entre ambos lenguajes y las dificultades que encontramos al realizar la traducción. Más tarde, propondremos un esquema de traducción que, aunque incompleto, puede servirnos para proporcionar al experto una versión inicial sobre la que trabajar.

3.2.1. Conocimiento estático en OWL-DL y PDDL

OWL-DL y PDDL son lenguajes creados con objetivos muy diferentes. OWL-DL se crea para representar conocimiento en la web semántica y, debido a la magnitud de este proyecto, su diseño presta especial atención a los problemas relativos a la adquisición y reutilización de las bases de conocimiento. OWL-DL representa el conocimiento usando ontologías que describen fragmentos de la realidad de manera reutilizable, y que se pueden combinar para representar fragmentos mayores. Además, prioriza la expresividad sobre la complejidad computacional de los procesos de inferencia, y asume que la información representada es incompleta. PDDL, por el contrario, se crea para poder comparar los algoritmos de búsqueda de varios planificadores en una competición. Los problemas de la competición se desarrollan en dominios con una estructura muy sencilla, por lo que PDDL asume una representación muy sencilla del vocabulario y del estado, y no presta atención a los problemas relativos al modelado de dominios complejos. De hecho, PDDL ha sido criticado como lenguaje por no tener un objetivo de diseño claro ni a nivel teórico ni a nivel práctico [87].

3.2.1.1. Mundo abierto y mundo cerrado

PDDL interpreta el conocimiento bajo la hipótesis de mundo cerrado (CWA), es decir, todo el conocimiento que no se proporciona de manera explícita ni puede deducirse a partir del conocimiento disponible se asume falso. Este enfoque simplifica enormemente la gestión de las bases de conocimiento pero limita el tipo de conocimiento que se puede expresar. En la versión determinista de PDDL el estado inicial es conocido, por lo que resulta natural usar CWA. En las versiones no deterministas de PDDL el estado inicial se describe como la disyunción de un número finito de estados posibles.

OWL-DL, sin embargo, interpreta el conocimiento usando la hipótesis de mundo abierto (OWA), donde no se asume nada sobre el conocimiento que no se tiene. Este enfoque permite representar de manera natural conocimiento incompleto sobre un dominio e ir completando la descripción del modelo con más información cuando se disponga de ella. Añadir nuevo conocimiento a una ontología OWL-DL no invalida ninguna de las inferencias que se hayan realizado previamente mientras la ontología actualizada siga siendo consistente.

Al traducir conocimiento de OWL-DL a PDDL añadimos de manera implícita información que *cierra* la interpretación y modifica la semántica original. Por ejemplo, el siguiente conjunto de átomos en OWL-DL describe una pizza que tiene al menos dos ingredientes (puede tener más), mientras que en PDDL se interpretaría como una pizza que tiene exactamente dos ingredientes. Estas diferencias de semántica pueden tener efectos inesperados: usando CWA la pizza p se clasificaría como pizza margarita mientras que usando OWA no tenemos suficiente información para afirmar si lo es o no.

$$Pizza(p), hasTopping(p, t1), hasTopping(p, t2),$$
$$Tomato(t1), Mozzarella(t2)$$

3.2.1.2. Nombre único o nombres múltiples

En PDDL dos constantes se consideran distintas por el mero hecho de usar nombres distintos. En OWL-DL, sin embargo, se pueden utilizar distintos nombres para hacer referencia a la misma entidad. Esta decisión de diseño se debe a que es frecuente encontrar ontologías creadas de manera independiente en las que se usan distintos nombres para representar la misma entidad. En OWL-DL, si queremos que dos nombres distintos (por ejemplo $t1$ y $t2$) se interpreten como entidades diferentes, debemos añadir dicha información de forma explícita ($t1 \neq t2$) o bien añadir información que permita a un razonador inferir que no son la misma entidad (por ejemplo indicando que $t1$ y $t2$ son instancias de clases disjuntas).

Por ejemplo, el siguiente conjunto de átomos interpretados en OWL-DL representan una pizza que tiene un ingrediente picante y ternera. La pizza podría tener un sólo ingrediente de tipo ternera picante, o dos ingredientes diferentes siendo uno de ellos picante y otro ternera, o incluso podría tener más de dos ingredientes. La interpretación de estos átomos en PDDL reduciría el conjunto de posible interpretaciones a una sola: una pizza con dos ingredientes.

$$Pizza(p), hasTopping(p, t1), hasTopping(p, t2),$$
$$Spicy(t1), Beef(t2)$$

3.2.1.3. Detectar inconsistencias en el estado

Los axiomas OWL-DL juegan un doble papel: en primer lugar establecen un conjunto de restricciones inviolables que permiten detectar inconsistencias; y en segundo lugar permiten inferir nuevo conocimiento que no se ha declarado explícitamente pero que es consecuencia lógica del conocimiento asertado. Estas dos visiones de los axiomas son complementarias e inseparables.

En PDDL no existen axiomas lógicos equivalentes a los de OWL-DL (ya veremos que los axiomas PDDL son distintos), pero sí podemos tratar de simular parcialmente su doble funcionalidad: inferir nuevo conocimiento y detectar inconsistencias. En este apartado nos centramos en la última parte, la detección de inconsistencias.

A partir de PDDL 3.0 se introduce la posibilidad de definir restricciones (*constraints*) que se deben cumplir durante la ejecución del plan. Usando este mecanismo podemos traducir los axiomas del dominio OWL-DL como restricciones que se deben cumplir *siempre*. Por ejemplo, el siguiente fragmento PDDL comprueba que las pizzas siempre tienen base, que ningún objeto puede ser de tamaño pequeño y mediano a la vez, y que siempre que usamos el predicado *hasTopping* relacionamos una pizza con un ingrediente.

```
(:constraints (and
   (always (forall (?x) (imply (Pizza ?x) (exists (?y)
                          (and (hasBase ?x ?y) (PizzaBase ?y))))))
   (always (not (exists (?x) (and (Small ?x) (Regular ?x)))))
   (always (forall (?x ?y) (imply (hasTopping ?x ?y)
                                  (and (Pizza ?x) (PizzaTopping ?y)))))
)
```

Usando este lenguaje de restricciones podemos detectar errores en la definición del estado inicial. También puede servirnos para detectar errores en la definición de los operadores si durante el proceso de planificación partimos de un estado consistente y llegamos a uno que no lo es.

En cualquier caso debemos tener en cuenta que la detección de inconsistencias en OWL-DL es mucho más potente, ya que no sólo podemos detectar inconsistencias en el estado (ABox) sino también en la definición del vocabuario del dominio (TBox).

3.2.1.4. Conceptos primitivos

La aproximación más sencilla para representar la jerarquía conceptual de la ontología es utilizar el sistema de tipos de PDDL. Desafortunadamente, el sistema de tipos de PDDL es extremadamente simple: cada tipo se representa mediante una etiqueta a la que no se puede asociar información semántica. La única relación entre los tipos es la relación *is-a* que permite definir una

taxonomía donde cada tipo sólo puede tener un padre en la jerarquía. Por ejemplo, la jerarquía de ingredientes de nuestra ontología quedaría así:

```
(:types PizzaTopping − Object
        Cheese, Meat, ..., Vegetable − PizzaTopping
        Mozzarella, GoatCheese, Gorgonzola, FourCheese − Cheese
        Ham, Beef, Chicken − Meat
        ...
)
```

Este enfoque tiene dos grandes problemas. El primero es que perdemos toda la información semántica asociada a cada tipo, ya que no podemos representar los axiomas que describen las relaciones entre los conceptos de la ontología. El segundo problema es que en PDDL cada tipo sólo puede tener un padre en la jerarquía, mientras que en las ontologías es común encontrar conceptos que especializan varios conceptos padres.

Otra aproximación más interesante, aunque no exenta de problemas, consiste en utilizar predicados para representar los conceptos de la ontología. Intuitivamente, los conceptos en la parte inferior de la ontología representan tipos concretos (*Mozzarella*, *Chicken*) mientras que los conceptos superiores representan tipos abstractos (*Vegetable*, *PizzaTopping*, ...). Podemos representar los tipos concretos utilizando predicados normales y los tipos abstractos utilizando *predicados derivados*. Los predicados derivados, también conocidos como *axiomas PDDL*, permiten definir predicados de alto nivel cuya interpretación depende de predicados normales y otros predicados derivados más simples. Por ejemplo, la jerarquía de ingredientes se representaría de la siguiente forma:

```
(:derived (PizzaTopping ?x)
          (or (Cheese ?x) (Meat ?x) ... (Vegetable ?x)))
(:derived (Cheese ?x)
          (or (Mozzarella ?x) (GoatCheese ?x) (Gorgonzola ?x)
              (FourCheese ?x)))
(:derived (Meat ?x)
          (or (Ham ?x) (Beef ?x) (Chicken ?x)))
         ...
)
```

Por ejemplo, el último predicado derivado indica que si en el estado existe un átomo *(Ham t)*, *(Beef t)* o *(Chicken t)* entonces también se cumple *(Meat t)*. Usando predicados derivados sí que podemos representar jerarquías complejas donde cada tipo tiene varios padres en la jerarquía (especializa varios tipos más generales).

Desafortunadamente este planteamiento también sufre ciertos problemas. Los axiomas que describen los conceptos primitivos tienen la forma $A \sqsubseteq C$ donde A es un concepto atómico y C un concepto cualquiera, posiblemente

complejo. Al representar estos axiomas como predicados derivados necesitamos representar C como uno o varios predicados derivados, lo cual sólo es posible cuando C es un concepto atómico o una conjunción de conceptos atómicos. Por ejemplo, los siguientes axiomas no podemos expresarlos usando predicados derivados:

$$MyFavoritePizza \sqsubseteq (>= 4 hasTopping) \sqcup (SpicyPizza) \quad (3.11)$$
$$Pizza \sqsubseteq hasBase.PizzaBase \quad (3.12)$$
$$(3.13)$$

3.2.1.5. Conceptos definidos

Los conceptos definidos son aquellos que aparecen a la izquierda de un axioma del tipo $A \equiv C$ donde A es un concepto atómico y C un concepto cualquiera, posiblemente complejo. Aunque estos axiomas se pueden utilizar para hacer inferencias en ambas direcciones ($A \sqsubseteq C$ y $C \sqsubseteq A$) lo habitual es querer que el razonador calcule automáticamente los subconceptos e instancias del concepto definido ($C \sqsubseteq A$). Si sólo queremos representar este sentido de la equivalencia podemos usar los predicados derivados tal y como hemos explicado en el apartado anterior. A continuación mostramos un par de conceptos definidos y su traducción a PDDL.

$$\begin{aligned} SpicyPizza &\equiv Pizza \sqcap hasTopping.SpicyTopping \\ Marguerita &\equiv Pizza \sqcap = 2 hasTopping \sqcap \\ &\quad \exists hasTopping.Cheese \sqcap \exists hasTopping.Tomato \end{aligned}$$

```
(:derived (SpicyPizza ?x)
    (and (Pizza ?x)
         (exists (?z1) (and (hasTopping ?x ?z1) (SpicyTopping ?z1)))))

(:derived (Margherita ?x)
    (and (Pizza ?x)
      (not (exists (?y1 ?y2 ?y3) (and (Different ?y1 ?y2 ?y3)
         (hasTopping ?x ?y1) (hasTopping ?x ?y1) (hasTopping ?x ?y1))))
      (exists (?z1) (and (hasTopping ?x ?z1) (Mozzarella ?z1)))
      (exists (?z2) (and (hasTopping ?x ?z2) (Tomato ?z2)))))
```

3.2.2. Esquema de traducción

A continuación describimos un algoritmo para traducir ontologías OWL-DL a PDDL usando todas las ideas que ya hemos visto. Volvemos a recordar que la traducción no mantiene la semántica original de la ontología: parte del conocimiento inicial se perderá y, además, añadiremos información que

no estaba en la ontología como consecuencia de *cerrar* la interpretación del mundo. Aún así, la traducción PDDL que obtendremos puede ser un buen punto de partida para el experto.

La traducción la haremos en 3 fases: traducción de la jerarquía conceptual usando predicados derivados, traducción de las restricciones del dominio usando *constraints*, y traducción del estado actual usando átomos. Si estamos usando la versión PDDL determinista el estado inicial debe ser único por lo que sólo debemos utilizar los conceptos hojas de la ontología para describirlo. A partir de estos átomos usaremos los predicados derivados para inferir los conceptos superiores de la ontología, y las restricciones para detectar estados inconsistentes.

Comenzamos traduciendo los conceptos primitivos. Como ya comentamos, sólo somos capaces de traducir los axiomas en los que la parte derecha es un concepto atómico o una conjunción de conceptos atómicos. Aunque esta aproximación es limitada, nos permite traducir la jerarquía básica. Sean A, B y B_i conceptos atómicos, los axiomas de inclusión de conceptos se traducen de la siguiente forma:

$$\begin{aligned} T_D(A \sqsubseteq B) &= (\textbf{:derived}\ (\text{B ?x})\ (\text{A ?x})) \\ T_D(A \sqsubseteq B_1 \sqcap \ldots \sqcap B_n) &= (\textbf{:derived}\ (B_i\ \text{?x})\ (\text{A ?x})) \end{aligned}$$

En realidad lo único que estamos diciendo es que a partir de predicados que se corresponden a las hojas de la ontología, podemos inferir los predicados que corresponden a los conceptos superiores.

A continuación traducimos los conceptos definidos. En este caso sólo nos quedamos con una de las dos direcciones de inferencia, la que permite calcular el concepto definido. Sea C un concepto cualquiera (posiblemente complejo), los axiomas de equivalencia de conceptos se traducen de la siguiente forma:

$$T_D(A \equiv C) = (\textbf{:derived}\ (\text{A ?x})\ \phi(?x, C))$$

donde $\phi(?x, C)$ representa la traducción del concepto C usando la función recursiva que se muestra en la tabla 3.4. El primer parámetro ($?x$) es necesario para indicar la variable libre que se usa en la fórmula de ámbito superior.

Una vez traducida la jerarquía conceptual pasamos a traducir las restricciones que nos permiten detectar estados inconsistentes. Sólo impondremos restricciones sobre los conceptos hoja de la ontología, ya que son los únicos que podemos utilizar para describir estados concretos.

$$T_C(A \sqsubseteq C) = (\textbf{always}\ (\textbf{forall}\ (\text{?x})\ (\textbf{imply}\ (\text{A ?x})\ \phi(?x, C))))$$

Además, añadimos restricciones sobre los dominios y rangos de las propiedades. Para cada propiedad R con dominio C y rango R añadimos la

$$
\begin{array}{rcl}
\phi(?x, A) & = & (\text{A ?x}) \\
\phi(?x, C \sqcap D) & = & (\textbf{and}\ \phi(?x, C)\ \phi(?x, D)) \\
\phi(?x, C \sqcup D) & = & (\textbf{or}\ \phi(?x, C)\ \phi(?x, D)) \\
\phi(?x, \neg C) & = & (\textbf{not}\ \phi(?x, C)) \\
\phi(?x, \exists R.C) & = & (\textbf{exists}\ (?\text{y})\ (\textbf{and}\ (\text{R ?x ?y})\ \phi(?y, C))) \\
\phi(?x, \forall R.C) & = & (\textbf{forall}\ (?\text{y})\ (\textbf{imply}\ (\text{R ?x ?y})\ \phi(?y, C))) \\
\phi(?x, \geq nR) & = & (\textbf{exists}\ (?y_1 \ldots ?y_n)\ (\textbf{and}\ (\text{Different } ?y_1 \ldots ?y_{n+1}) \\
 & & \qquad (\text{R ?x } ?y_1) \ldots (\text{R ?x } ?y_{n+1}))) \\
\phi(?x, \leq nR) & = & \phi(?x, \neg(\geq (n+1)R)) \\
\phi(?x, \{a\}) & = & (= \text{?x a})
\end{array}
$$

Tabla 3.4: Traducción de formulas OWL-DL a PDDL.

siguiente restricción:

$$
\begin{array}{rcl}
T_C(R : C \rightarrow D) & = & (\textbf{always}\ (\textbf{forall}\ (\text{?x ?y})\ (\textbf{imply}\ (\text{R ?x ?y}) \\
 & & \qquad (\textbf{and}\ \phi(?x, C)\ \phi(?y, D))))
\end{array}
$$

Finalmente traducimos los axiomas de la ABox que representan el estado de la siguiente forma:

$$
\begin{array}{rcl}
T_S(C(a)) & = & (\text{C a}) \\
T_S(R(a, b)) & = & (\text{R a b})
\end{array}
$$

3.2.3. Ejemplo

A continuación mostramos un pequeño ejemplo de cómo funciona la traducción. Partiremos del siguiente conjunto de axiomas, que representa una pequeña parte de la ontología sobre la pizzería:

$$
\begin{array}{rcl}
Gongonzola & \sqsubseteq & Cheese \\
Cheese & \sqsubseteq & PizzaTopping \\
SlicedTomato & \sqsubseteq & Tomato \\
Tomato & \sqsubseteq & Vegetable \\
Vegetable & \sqsubseteq & PizzaTopping \\
Vegetable & \sqsubseteq & \neg Cheese \\
VegetarianTopping & \equiv & Cheese \sqcup Fruit \sqcup HerbSpice \sqcup \\
 & & Nut \sqcup Sauce \sqcup Vegetable \\
VegetarianPizza & \equiv & Pizza \sqcap \forall hasTopping.VegetarianTopping \\
hasTopping & : & Pizza \rightarrow PizzaTopping
\end{array}
$$

Además, supondremos que la ontología cuenta con los siguientes axiomas en la ABox para describir una situación concreta:

```
(define (domain DX)
   (:requirements :derived−predicates :constraints)

   (:predicates (Gongonzola ?x) (Cheese ?x) (PizzaTopping ?x)
                (SlicedTomato ?x) (Tomato ?x) (Vegetable ?x)
                (VegetarianTopping ?x) (Fruit ?x) (HerbSpice ?x) (Nut ?x)
                (Sauce ?x) (VegetarianPizza ?x) (hasTopping ?x ?y))

   (:derived (Cheese ?x) (Gongonzola ?x))
   (:derived (PizzaTopping ?x) (Cheese ?x))
   (:derived (Tomato ?x) (SlicedTomato ?x))
   (:derived (Vegetable ?x) (Tomato ?x))
   (:derived (PizzaTopping ?x) (Vegetable ?x))
   (:derived (VegetarianTopping ?x) (or (Cheese ?x) (Fruit ?x) (HerbSpice ?x)
                (Nut ?x) (Sauce ?x) (Vegetable ?x)))
   (:derived (VegetarianPizza ?x) (and (Pizza ?x) (forall (?y)
                (imply (hasTopping ?x ?y) (VegetarianTopping ?y)))))

   (:constraints (and
      (always (forall (?x) (imply (Vegetable ?x) (not (Cheese ?x)))))
      (always (forall (?x ?y) (imply (hasTopping ?x ?y)
                                     (and (Pizza ?x) (PizzaTopping ?y))))))
   ))  ...)

(define (problem PD)
   (:domain DX)
   (:objects t1 t2 b1 p1)
   (:init  (Gongonzola t1) (SlicedTomato t2)
           (PizzaBase b1) (Pizza p1)
           (hasTopping p1 t1) (hasTopping p1 t2)) ...)
```

Figura 3.4: Ejemplo de traducción a PDDL

$$Gongonzola(t1), SlicedTomato(t2), PizzaBase(b1), Pizza(p1),$$
$$hasTopping(p1, t1), hasTopping(p1, t2)$$

Usando el esquema de traducción del apartado anterior conseguiríamos una descripción parcial de dominio y problema PDDL como la que se muestra en la figura 3.4.

3.3. Planificación usando OWL-DL

En el siguiente capítulo exploraremos otra aproximación totalmente distinta. En lugar de traducir las ontologías del dominio al lenguaje PDDL, con los problemas que ello conlleva, estudiaremos la posibilidad de resolver

problemas de planificación usando lógicas descriptivas para representar el conocimiento. Al utilizar un lenguaje más expresivo, el proceso de planificación se complica, pero a cambio podemos utilizar las ontologías del dominio con su semántica original. Las principales ventajas de este nuevo enfoque son:

- El uso de ontologías facilita el proceso de modelado de dominios ricos en conocimiento. Podemos definir dominios de manera modular y extender otras ontologías existentes. Además disponemos de potentes herramientas que nos ayudarán durante la formalización.
- Las ontologías proporcionan un vocabulario expresivo con el que definir estados iniciales y operadores de planificación. La capacidad de razonar a partir de descripciones incompletas promueve la reutilización de los planes generados y abre las puertas a aproximaciones basadas en casos.
- Los razonadores DL están optimizados para gestionar grandes cantidades de información. Estos sistemas implementan técnicas especializadas para razonar de manera incremental y responder consultas de manera eficiente.
- Las capacidades de inferencia de los razonadores DL son una potente herramienta para realizar interesantes inferencias durante el proceso de búsqueda de planes.
- Además, podemos detectar automáticamente inconsistencias en la base de conocimiento y ayudar al diseñador a corregir los errores mostrando los conjuntos de axiomas que entran en conflicto (técnicas para depurar ontologías [71]).
- El razonamiento en mundo abierto nos permite representar el espacio de búsqueda de manera compacta. Al representar el estado como una base de conocimiento DL que admite muchos modelos (incluso infinitos) estamos representando muchos posibles estados de una vez.
- Las ontologías suelen contener mucho conocimiento sobre el dominio que se podría tratar de explotar para guiar la búsqueda de manera efectiva.

3.4. Conclusiones

En este capítulo hemos propuesto el uso de ontologías para representar el conocimiento estático de los dominios de planificación. Las ontologías facilitan la creación de modelos ricos en conocimiento gracias a los lenguajes expresivos con los que se describen y las herramientas que nos facilitan su gestión (editores, razonadores, repositorios, ...). Como ejemplo hemos visto

una ontología que modela una pizzería en la que se describen semánticamente distintos tipos de ingredientes, bases y pizzas. El vocabulario que hemos descrito en esta ontología es extremadamente simple si lo comparamos con otras ontologías de la web semántica, pero es muy complejo si lo comparamos con los vocabularios que se suelen usar en los dominios de planificación.

Después hemos visto los problemas que surgen cuando intentamos traducir ontologías OWL-DL a PDDL debido a las diferencias entre ambos lenguajes. Aun así, hemos propuesto un algoritmo que traduce de manera parcial el conocimiento de las ontologías usando predicados derivados y restricciones PDDL. El resultado de esta traducción debe ser revisado por un experto que decidirá qué conocimiento es necesario y cual es la mejor manera de representarlo. Además, deberá completar la definición del dominio con los operadores de planificación adecuados. Aún así, el algoritmo de traducción que hemos propuesto puede proporcionar una buena base sobre la que empezar a trabajar.

Finalmente hemos planteado una nueva aproximación que desarrollaremos en los siguientes capítulos: planificar usando lógicas descriptivas para representar el conocimiento y, de ese modo, aprovechar toda la expresividad de estos formalismos. Aunque la planificación con DLs tiene evidentes ventajas desde el punto de vista de la expresividad, veremos que, hoy por hoy, también tiene muchos problemas desde el punto de vista práctico. En cualquier caso, es una aproximación que puede resultar muy interesante en problemas en los que la solución es trivial si se sabe gestionar de manera adecuada el conocimiento estático del dominio. Nos referimos a problemas en los que los planes solución tienen pocas acciones pero para hallarlas en necesario realizar inferencias complejas a partir del conocimiento disponible.

Capítulo 4

Planificación usando lógicas descriptivas

En el capítulo anterior hemos visto cómo podemos utilizar DLs para modelar de manera natural el conocimiento estático de un dominio. También describimos las dificultades que surgen cuando intentamos traducir todo este conocimiento al lenguaje estándar de planificación. En este capítulo abordamos una aproximación diferente: en lugar de traducir el conocimiento para usar un planificador convencional, describimos un modelo de planificación que usa DLs para representar el conocimiento. Esto nos permitirá utilizar las ontologías que hemos creado para describir dominios con vocabulario complejos usando su semántica original.

La capacidad expresiva de las DLs será útil para describir dos tipos de conocimiento: (1) el vocabulario del dominio, y de ese modo poder representar dominios estructurados con un gran número de restricciones; y (2) el estado del sistema, lo que nos permitirá trabajar con información incompleta y representar de manera compacta el espacio de búsqueda. Además, usando DLs podremos detectar automáticamente inconsistencias en el conocimiento representado y realizar inferencias interesantes durante el proceso de búsqueda.

La posibilidad de trabajar con información incompleta es especialmente interesante ya que permite plantear problemas en los que el estado inicial se describe de forma abstracta, sin proporcionar ciertos detalles que no se conocen o no interesan. Al resolver uno de estos problemas abstractos, lo que estamos haciendo en realidad es buscar soluciones válidas para muchos problemas concretos (todos aquellos más específicos que el problema abstracto original). Los planes solución, en estos casos, se pueden ver como estrategias que permiten resolver familias de problemas. A lo largo de este capítulo desarrollamos esta idea, que volveremos a retomar en el capítulo 6 para definir un planificador basado en casos que explota la capacidad de las DLs para representar conocimiento a distintos niveles de abstracción.

Pero para ser capaces de describir dominios y problemas de planificación no es suficiente con modelar la parte de conocimiento estática, también necesitamos expresar las acciones que un agente puede realizar. En este capítulo veremos cómo describir estas acciones y sus consecuencias usando el vocabulario de la ontología del dominio, y algunos de los problemas relativos a su interpretación.

Comenzaremos el capítulo planteando el problema de planificación como una búsqueda en el espacios de estados. A partir de una descripción parcial del estado inicial del sistema y de los objetivos del problema, buscaremos secuencias de acciones que garanticen el cumplimiento de dichos objetivos. Esta representación del problema como búsqueda en un grafo de estados resulta sencilla de entender, pero sólo resulta eficaz si disponemos de buenas heurísticas que permitan reducir de manera efectiva el espacio de búsqueda.

A continuación plantearemos el problema desde el punto de vista de la planificación jerárquica. Sólo podremos utilizar esta aproximación si disponemos de conocimiento del dominio adicional que nos permita modelar los problemas como una búsqueda en el espacio de descomposición de tareas. El objetivo de un problema, en este caso, consiste en resolver una tarea o secuencia de tareas, y para hacerlo las descomponemos en subtareas más sencillas de manera recursiva hasta alcanzar tareas primitivas que se pueden resolver usando operadores. La ventaja de esta aproximación es que, gracias a que disponemos de mucha información para guiar la búsqueda, podremos resolver problemas de manera más eficiente.

4.1. Búsqueda en el espacio de estados

Una base de conocimiento DL consta de dos partes bien diferenciadas: la parte terminológica (TBox) y la parte asertiva (ABox). La parte terminológica representa el vocabulario y los axiomas del dominio, es decir, conocimiento que tiende a permanecer sin cambios durante la vida de la aplicación. La parte asertiva, sin embargo, se suele utilizar para representar información más volátil, como por ejemplo el estado del sistema en un determinado instante de tiempo.

Nosotros vamos a utilizar la misma aproximación. Representaremos todo el conocimiento estático de planificación en una base de conocimiento DL donde la parte terminológica describe el vocabulario del dominio y la parte asertiva describe el estado del planificador. Las acciones de planificación, por tanto, modificarán el estado del sistema realizando cambios en la ABox.

Definición 1 (Estado) *Un estado es una base de conocimiento $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ siendo $\mathcal{T}$ una TBox acíclica. Decimos que el estado es consistente si la base de conocimiento $\mathcal{K}$ es consistente, e inconsistente en caso contrario.*

En realidad, la base de conocimiento (KB en adelante) es una descripción

parcial e incompleta del estado real del sistema, que es desconocido. La KB puede admitir varios modelos y sólo uno de ellos corresponde al estado real, pero no sabemos cual. Esta capacidad de las DLs para razonar con conocimiento incompleto nos será útil para definir problemas de planificación en los que el estado inicial se describe de forma abstracta. Los planes solución deben ser válidos para cualquier estado inicial concreto compatible con la descripción parcial que proporcionamos.

A partir de ahora supondremos que la KB que describe el estado del sistema es siempre consistente. En caso contrario debemos alertar al usuario, ya una KB inconsistente implica que existe algún error en el modelo del dominio o del problema de planificación.

4.1.1. Acciones

Las acciones representan las distintas maneras que tiene un agente de interactuar con el entorno. Cada acción se describe de manera declarativa mediante una *precondición* (condiciones necesarias para poder ejecutarla) y una *postcondición* (efectos causados tras su ejecución). En realidad, la postcondición de la acción sólo proporciona una descripción parcial del estado que se alcanzará tras ejecutarla. La manera concreta de alcanzar ese estado, es decir, los cambios concretos que debemos realizar en el estado original, dependen de la semántica concreta que elijamos.

Describir acciones puede ser problemático, sobre todo cuando existe una teoría del dominio que relaciona los predicados mediante axiomas. Los tres problemas más estudiados en la literatura son: *the frame problem* [85], que consiste en caracterizar qué parte del estado no cambia tras ejecutar la acción; *the ramification problem* [41], que consiste en caracterizar los efectos indirectos producidos por la acción; y *the qualification problem* [84], que consiste en caracterizar correctamente las precondiciones de las acciones.

Antes de describir las acciones que vamos a utilizar necesitamos introducir el concepto de literal primitivo.

Definición 2 (Literal primitivo) *Sea N_I un conjunto no vacío de nombres de individuos y $\mathcal{T}$ una TBox acíclica. Un literal primitivo de $\mathcal{T}$ es un axioma de la forma $A(a)$, $\neg A(a)$, $r(a,b)$, $\neg r(a.b)$ donde A es un concepto primitivo de $\mathcal{T}$, r un nombre de rol, y $a, b \in N_I$ nombres de individuos.*

A continuación definimos la sintaxis y semántica de nuestras acciones. Exigimos que la TBox que define el vocabulario del dominio sea acíclica y que no existan conceptos definidos en la postcondición para evitar problemas a la hora de definir su semántica [8].

Definición 3 (Acción) *Sea N_I un conjunto no vacío de nombres de individuos y $\mathcal{T}$ una TBox acíclica. Una acción $\alpha = (pre, post)$ de $\mathcal{T}$ es un vector donde:*

- *pre, la precondición, es un conjunto finito de axiomas ABox de* $\mathcal{T}$.
- *post, la postcondición, es un conjunto finito de literales primitivos de* $\mathcal{T}$.

Definición 4 (Transición entre modelos) *Sea* $\mathcal{T}$ *una TBox acíclica,* $\alpha = (pre, post)$ *una acción de* $\mathcal{T}$, $\mathcal{I}$, $\mathcal{I}'$ *modelos de* $\mathcal{T}$ *que respetan la hipótesis de nombre único y que comparten el mismo dominio e interpretación para todos los nombres de individuos. Decimos que la acción* α *puede transformar* $\mathcal{I}$ *en* $\mathcal{I}'$ *(*$\mathcal{I} \Rightarrow_\alpha^\mathcal{T} \mathcal{I}'$*) sii para cada concepto primitivo* A *y role* r*:*

$$\begin{aligned} A^{\mathcal{I}'} &:= (A^\mathcal{I} \cup \{a^\mathcal{I} \mid A(a) \in post\}) \setminus \{a^\mathcal{I} \mid \neg A(a) \in post\} \\ r^{\mathcal{I}'} &:= (r^\mathcal{I} \cup \{(a^\mathcal{I}, b^\mathcal{I}) \mid r(a,b) \in post\}) \setminus \\ &\qquad \{(a^\mathcal{I}, b^\mathcal{I}) \mid \neg r(a,b) \in post\} \end{aligned}$$

Existen varias precisiones importantes que debemos realizar. En primer lugar, definimos la semántica de la acción en función de transformación de modelos concretos de un KB. Esta forma de trabajar nos permite independizarnos de la representación sintáctica de la KB que estemos usando. Dos KBs sintácticamente distintas pero semánticamente equivalentes tendrán los mismos modelos por lo que la ejecución de una acción tendrá las mismas consecuencias en las dos. En segundo lugar, nuestra definición no tiene en cuenta si la acción es ejecutable o no, es decir, si sus precondiciones se satisfacen. Sólo indicamos el resultado de ejecutar la acción independientemente de que se pueda ejecutar o no. Además, como suponemos la TBox acíclica, la interpretación de los conceptos definidos queda determinada por la interpretación de los conceptos primitivos y nombres de roles, y por tanto no es necesario considerar los conceptos definidos cuando describimos la relación de transición. Finalmente, podemos considerar las acciones deterministas en el sentido de que, como la TBox debe ser acíclica, no puede existir más de una interpretación $\mathcal{I}'$ tal que $\mathcal{I} \Rightarrow_\alpha^\mathcal{T} \mathcal{I}'$.

Sin embargo, aunque las acciones sean deterministas, debemos tener en cuenta que desde el punto de vista del planificador sí que existe indeterminismo. La KB que representa el estado puede admitir varios modelos, por lo que tras ejecutar una acción llegaremos a otra KB que, seguramente, también admita varios modelos.

Definición 5 (Transición entre estados) *Sea* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *el estado actual del sistema,* $\alpha = (pre, post)$ *una acción de* $\mathcal{T}$*, y* $\mathcal{M}(\mathcal{K})$ *el conjunto de todos los modelos de* $\mathcal{K}$*. Decimos que* α *es aplicable en el estado* $\mathcal{K}$ *si para todo modelo* $\mathcal{I} \in \mathcal{M}(\mathcal{K})$ *se cumple que* $\mathcal{I} \models pre$*.*

El resultado de aplicar α *en* $\mathcal{K}$*, denotado por* $\gamma(\mathcal{K}, \alpha)$*, es otro estado* $\mathcal{K}' = (\mathcal{T}, \mathcal{A}')$ *tal que* $\mathcal{M}(\mathcal{K}') = \{\mathcal{I}' \mid \mathcal{I} \Rightarrow_\alpha^\mathcal{T} \mathcal{I}', \mathcal{I} \in \mathcal{M}(\mathcal{K})\}$.

A continuación extendemos la idea de acción como trasformadora de modelos a secuencias de acciones.

Definición 6 (Transición entre modelos - secuencias) *Decimos que la secuencia de acciones* $\langle \alpha_1 \dots \alpha_k \rangle$ *puede transformar* $\mathcal{I}$ *en* $\mathcal{I}'$ ($\mathcal{I} \Rightarrow^{\mathcal{T}}_{\langle \alpha_1 \dots \alpha_k \rangle} \mathcal{I}'$) *sii existen modelos* $\mathcal{I}_0 \dots \mathcal{I}_k$ *de* $\mathcal{T}$ *tales que* $\mathcal{I} = \mathcal{I}_0$, $\mathcal{I}' = \mathcal{I}_k$ *y* $\mathcal{I}_{i-1} \Rightarrow^{\mathcal{T}}_{\alpha_i} \mathcal{I}_i$ *para* $1 \leq i \leq k$.

Finalmente, podemos definir cuándo una secuencia de acciones es ejecutable a partir de un cierto estado. Para poder garantizarlo debemos comprobar que cada acción se puede ejecutar en todos los modelos alcanzables usando las acciones anteriores.

Definición 7 (Ejecución) *Sea* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *un estado y* $\langle \alpha_1 \dots \alpha_n \rangle$ *una secuencia de acciones de* $\mathcal{T}$ *tales que* $\alpha_i = (pre_i, post_i)$ *para* $i = 1 \dots n$. *Decimos que la secuencia de acciones* $\langle \alpha_1 \dots \alpha_n \rangle$ *es ejecutable en* $\mathcal{K}$ *sii las condiciones siguientes se cumplen para todos los modelos* $\mathcal{I}$ *de* $\mathcal{K}$:

- $\mathcal{I} \models pre_1$
- *para todo* i *tal que* $1 \leq i \leq n$ *y para todas las interpretaciones* $\mathcal{I}'$ *tales que* $\mathcal{I} \Rightarrow^{\mathcal{T}}_{\langle \alpha_1 \dots \alpha_i \rangle} \mathcal{I}'$ *se cumple que* $\mathcal{I}' \models pre_{i+1}$.

Relacionado con el problema anterior está el problema de la proyección, que consiste en determinar si podemos garantizar que se cumplen unas ciertas condiciones tras ejecutar una secuencia de acciones. Igual que antes, para poder garantizarlo necesitamos comprobar que dichas condiciones se cumplen en cualquiera de los modelos potencialmente alcanzables.

Definición 8 (Proyección) *Decimos que una sentencia* φ *es consecuencia de aplicar la secuencia de acciones* $\langle \alpha_1 \dots \alpha_n \rangle$ *a partir del estado* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *sii para todos los modelos* $\mathcal{I}$ *de* $\mathcal{K}$ *y para todo* $\mathcal{I}'$ *tal que* $\mathcal{I} \Rightarrow^{\mathcal{T}}_{\langle \alpha_1 \dots \alpha_n \rangle} \mathcal{I}'$ *se cumple que* $\mathcal{I}' \models \varphi$.

4.1.2. Dominios y problemas de planificación

Aunque las acciones son los elementos esenciales con los que trabaja un planificador, la descripción de un dominio de planificación no se suele hacer usando acciones sino operadores. Los operadores son acciones parametrizadas, es decir, acciones que pueden utilizar variables en lugar de nombres de individuos.

La precondición de los operadores pasa a ser, por tanto, una consulta conjuntiva DL. Al ejecutar la consulta sobre la base de conocimiento que representa el estado actual del sistema obtendremos sustituciones que vincularán las variables del operador a individuos concretos. La idea intuitiva

es que usando esas sustituciones podemos transformar el operador en una acción concreta y ejecutarla.

Recordemos que las consultas DL tienen la forma $Q(\mathbf{x}, \mathbf{y})$ siendo $\mathbf{x}$ el conjunto de variables distinguibles e $\mathbf{y}$ el conjunto de variables no distinguibles. El resultado de efectuar una consulta $Q(\mathbf{x}, \mathbf{y})$ sobre una base de conocimiento $\mathcal{K}$ es un conjunto de sustituciones σ que vincula las variables distinguibles de la consulta a individuos de $\mathcal{K}$ de manera que la consulta $Q(\mathbf{x}[\sigma], \mathbf{y})$ es una consecuencia lógica de $\mathcal{K}$ (sección 2.1.3.5).

Nuestros operadores sólo utilizarán variables distinguibles. Esta simplificación nos permite mantener la idea intuitiva de poder transformar los operadores en acciones simplemente aplicándoles las sustituciónes que se obtienen tras evaluar la precondición. Sería fácil, sin embargo, extender la definición para permitir el uso de variables no distinguibles en la precondición.

Definición 9 (Operador de planificación) *Sean N_I, N_X conjuntos disjuntos no vacíos de nombres de individuos y variables respectivamente, y sea $\mathcal{T}$ una TBox acíclica. Una operador de planificación $o = (name, pre, post)$ es un vector donde:*

- *name, el nombre del operador, es una expresión sintáctica de la forma $n(x_1 \dots x_k)$ donde n es un símbolo de operador y $x_1 \dots x_k \in N_X$ son todos los símbolos de variable del operador.*
- *$pre(\mathbf{x}, \emptyset)$, la precondición, es una consulta DL que usa el vocabulario de de $\mathcal{T}$.*
- *$post(\mathbf{y}, \emptyset)$, la postcondición, es otra consulta DL que usa el vocabulario de $\mathcal{T}$ y que sólo usa nombres de conceptos primitivos. Además $\mathbf{y} \subseteq \mathbf{x}$.*

Sea $\sigma : \{x_1 \dots x_k\} \to N_I$ una sustitución que asocia individuos a cada variable. Usaremos $\alpha := o[\sigma]$ para denotar la acción resultante de aplicar la sustitución σ al operador o.

Llegados a este punto podemos definir los conceptos de modelo del dominio y modelo del problema de planificación. El dominio contiene la descripción del vocabulario y un conjunto de operadores. Un problema de planificación completa la información del dominio definiendo el estado inicial y los objetivos del problema.

Definición 10 (Dominio de planificación) *Un dominio de planificación es un vector $\mathcal{D} = (\mathcal{T}, \mathcal{O})$ donde $\mathcal{T}$ es una TBox acíclica, y $\mathcal{O} = \{o_1 \dots o_n\}$ un conjunto de operadores de planificación.*

Definición 11 (Problema de planificación) *Un problema de planificación $\mathcal{P} = (\mathcal{T}, \mathcal{A}, \mathcal{O}, G)$ es un vector donde:*

- $\mathcal{D} = (\mathcal{T}, \mathcal{O})$ *es un dominio de planificación.*
- $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *es el estado inicial del problema.*
- G *es una consulta conjuntiva booleana DL que se satisface en los estados objetivos.*

El objetivo del planificador es encontrar un plan, es decir, una secuencia de acciones (operadores y sustituciones) que garanticen la consecución de los objetivos del problema.

Definición 12 (Plan) *Un plan* $\pi = \langle \alpha_1 \dots \alpha_n \rangle$ *es una secuencia de acciones. Decimos que el plan* π *es una solución al problema de planificación* $\mathcal{P} = (\mathcal{T}, \mathcal{A}, \mathcal{O}, G)$ *si:*

- π *es ejecutable a partir de* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$.
- Q *es consecuencia de aplicar* π *en* $\mathcal{K}$.
- *cada* $\alpha_i = o_i[\sigma_i]$ *para algún* $o_i \in \mathcal{O}$ *y alguna sustitución* σ_i.

4.1.3. Ejemplo

A continuación mostramos un ejemplo que utiliza la ontología de la pizzería que describimos en el capítulo anterior. Esta ontología nos proporciona el vocabulario necesario para describir problemas en los que debemos combinar ingredientes para crear distintos tipos de pizzas. Los dos operadores que vamos a utilizar en este ejemplo los describimos a continuación:

operator addTopping(?p,?t)
pre: NonCookedPizza(?p), AvailableTopping(?t)
post: not AvailableTopping(?t), hasTopping(?p,?t)

operator bakePizza(?p)
pre: NonCookedPizza(?p)
post: CookedPizza(?p)

El operador *addTopping* permite añadir un ingrediente a una pizza. La precondición exige que la pizza esté sin cocinar y que el ingrediente esté disponible (no haya sido añadido ya a otra pizza). La postcondición indica que el ingrediente ya no estará disponible y que se habrá añadido a la pizza. El segundo operador, *bakePizza*, sirve para cocinar una pizza. En este caso la precondición selecciona una pizza sin cocinar y la postcondición la transforma en una pizza cocinada.

Nuestro dominio de planificación queda definido usando la ontología de la pizzería y los operadores anteriores. Para describir un problema de planificación en ese dominio necesitamos definir el estado inicial y el objetivo a alcanzar. El estado inicial de nuestro problema es el siguiente:

$$VegetarianPizza(p), NonCookedPizza(p), Fish(t1),$$
$$Beef(t2), AvailableTopping(t2)$$

Este estado lo único que indica es que existe una pizza p vegetariana y sin cocinar, un ingrediente $t1$ que es algún tipo de pescado, y otro ingrediente $t2$ de tipo ternera que aún no ha sido usado en ninguna pizza. Existen multitud de de estados concretos compatibles con esta descripción parcial: la pizza p puede tener cualquier número de ingredientes siempre que sean de tipo vegetariano, el ingrediente $t1$ puede ser cualquier tipo de pescado y puede estar disponible o formar parte de alguna otra pizza, incluso puede haber más pizzas e ingredientes de los que no decimos nada.

Como objetivo del problema pretendemos conseguir una pizza con carne cocinada:

$$MeatyPizza(?x), CookedPizza(?x)$$

Antes de resolver el problema pensemos un poco sobre él. Lo que pretendemos es transformar una pizza vegetariana sin cocinar en una pizza con carne y cocinada. Intuitivamente resolver este problema es muy sencillo, lo único que debemos hacer es añadir cualquier ingrediente disponible de tipo carne y luego cocinar la pizza. La dificultad de este problema no está en la longitud del plan solución, sino en la gestión del conocimiento del dominio.

El dominio de las pizzas está descrito mediante una base de conocimiento que contiene cientos de axiomas. Además, la descripción del estado inicial de este problema es muy abstracta. De hecho, debido a la interpretación del conocimiento del mundo abierto, existen infinitos posibles estados iniciales que encajan con nuestra descripción parcial (la pizza vegetariana podría tener cualquier número de ingredientes). Al resolver este problema, estamos resolviendo infinitos problemas de una vez. Es más, sin mundo abierto no podríamos plantear el problema general de trasformar una pizza vegetariana en una pizza con carne.

A este planteamiento se podría objetar que, a efectos prácticos, las pizzas sólo pueden tener un número limitado de ingredientes y que el número de pizzas vegetarianas con n ingredientes sí es finito. Incluso aceptando este argumento, basta echar algunas cuentas para ver que el espacio de posibles estados sigue siendo muy grande. En nuestra ontología tenemos 37 ingredientes distintos (29 vegetarianos), 3 tamaños de pizzas y 2 tipos de bases, así que suponiendo pizzas con 8 o menos ingredientes en los que no se repita ninguno podemos crear 310,4 millones de pizzas diferentes de las cuales 38,8 milllones son vegetarianas. Por lo tanto, incluso cerrando la interpretación del mundo estamos intentando buscar un plan válido para más de 38 millones de problemas concretos. Y sin embargo, el problema que hemos planteado es

trivial de resolver si se entienden los conceptos implicados (pizza vegetariana, pizza con carne, añadir ingrediente, concinar pizza, ...). Para este tipo de problemas sí puede resultar interesante usar DLs, problemas cuya dificultad se basa en la gestión del conocimiento del dominio y no tanto en la explosión combinatoria del espacio de búsqueda que surge al aumentar el tamaño del plan solución.

Volviendo al problema concreto que nos ocupa, partimos de un estado inicial en el que tenemos una pizza p vegetariana y no cocinada, un ingrediente $t1$ de tipo pescado, y un ingrediente disponible $t2$ de tipo ternera. En este estado podemos ejecutar la acción $addTopping(p, t2)$ ya que su precondición se cumple de manera trivial. El resultado que obtendremos será un nuevo estado donde p representa una pizza sin cocinar que contiene ternera y cualquier otro número de ingredientes vegetarianos. Escribir este tipo de estados sin alterar la TBox de nuestra base de conocimiento es problemático. Al final de este capítulo profundizaremos en este problema, por ahora nos quedaremos con la idea intuitiva de que para describir estos estados necesitamos ampliar un poco el lenguaje (usando con el constructor "@" de la lógica híbrida y nominales, o Boolean ABoxes [82]).

Por ahora esquivaremos este problema describiendo el estado siguiente con una condición más débil de la que realmente alcanzaríamos: p es una pizza no cocinada que contiene ternera. Como estamos perdiendo información sobre el estado es posible que no podamos calcular todas las soluciones del problema, pero en ningún caso esta pérdida de información puede invalidar las soluciones que sí encontremos.

$$NonCookedPizza(p), hasTopping(p, t2), Fish(t1), Beef(t2)$$

Finalmente aplicamos la acción $bakePizza(p)$ y obtenemos el siguiente estado:

$$CookedPizza(p), hasTopping(p, t2), Fish(t1), Beef(t2)$$

Este estado cumple con el objetivo del problema ya que p es una pizza con ternera y por tanto es una pizza con carne.

4.2. Planificación STN

Como comentamos al principio del capítulo, si disponemos de suficiente información sobre el dominio, podemos resolver problemas de manera muy eficiente usando planificación jerárquica. En concreto nosotros vamos a describir un modelo conocido como planificación STN (*Simple Task Network*) que es una versión simplificada del modelo HTN ([100] capítulo 11) .

En este tipo de planificación el problema se describe mediante un conjunto de tareas a realizar. Las tareas se van descomponiendo por medio de métodos en subtareas más sencillas, hasta alcanzar tareas simples o primitivas que se pueden resolver usando operadores. De este modo, el proceso de búsqueda se puede representar por medio de un árbol de descomposición de tareas cuyas hojas son las acciones que componen el plan solución.

4.2.1. Tareas, métodos y operadores

Comenzaremos definiendo las tareas. A diferencia del modelo STN clásico nosotros no haremos distinción entre tareas primitivas y no primitivas, es decir, podemos definir varios operadores y métodos que resuelven la misma tarea de formas distintas. De este modo separamos completamente el concepto de la tarea, como objetivo que debemos alcanzar, del modo de resolución empleado.

Definición 13 (Tarea) *Una tarea es un vector* $t = (s, p_1, \ldots, p_n)$ *donde* s *es el símbolo de la tarea y* $p_1 \ldots p_n$ *son nombres de individuos o nombres de variables.*

La representación de las tareas en nuestro modelo es muy sencilla. Como no representamos de manera explícita su semántica, cada operador y método tendrá que describir de forma explícita la tarea que es capaz de resolver. Existen otras aproximaciones que enriquecen las descripciones de tareas mediante precondiciones y postcondiciones o incluso las representan de manera jerárquica usando ontologías [123]. Esta información extra permite recuperar los métodos y operadores que saben resolver una tarea sin necesidad de representar este vínculo de manera explícita, usando operaciones de subsunción entre consultas DL. Nosotros, para simplificar, supondremos que cada operador y método define la tarea que puede resolver.

A continuación definimos el concepto de red de tareas como un conjunto de tareas y una relación de orden (parcial o total) que indica el orden en el que deben resolverse.

Definición 14 (Red de tareas) *Una red de tareas es un grafo acíclico* $w = (U, E)$ *donde* U *es un conjunto de nodos,* E *es un conjunto de aristas, y cada nodo* $u \in U$ *contiene una tarea* t_u*.*

Las aristas de w *definen un orden parcial sobre* U*, en concreto* $u \prec v$ *si hay un camino de* u *a* v*. Si el orden parcial es total decimos que* w *es una red de tareas totalmente ordenada.*

Los operadores STN son equivalentes a los operadores clásicos que vimos en la primera parte del capítulo, pero contienen información adicional sobre la tarea que resuelven. Cada operador indica la tarea que puede resolver, las

condiciones que se deben dar en el estado del mundo para ser aplicable y los efectos de su aplicación.

Definición 15 (Operador STN) *Sean N_I, N_X conjuntos disjuntos no vacíos de nombres de individuos y variables respectivamente, y sea $\mathcal{T}$ una TBox acíclica. Un operador STN $o = (name, task, pre, post)$ es un vector donde:*

- *name, el nombre del operador, es una expresión sintáctica de la forma $n(x_1 \dots x_k)$ donde n es un símbolo de operador y $x_1 \dots x_k \in N_X$ son todos los símbolos de variable del operador.*
- *task es la tarea que este operador es capaz de resolver.*
- *$pre(\mathbf{x}, \emptyset)$, la precondición, es una consulta DL que usa el vocabulario de $\mathcal{T}$.*
- *$post(\mathbf{y}, \emptyset)$, la postcondición, es otra consulta DL que usa el vocabulario de $\mathcal{T}$ y sólo usa nombres de conceptos primitivos. Además $\mathbf{y} \subseteq \mathbf{x}$.*

Sea $\sigma : \{x_1 \dots x_k\} \to N_I$ una sustitución que asocia individuos a cada variable. Usaremos $\alpha := o[\sigma]$ para denotar la acción resultante de aplicar la sustitución σ al operador o.

Los métodos permiten resolver tareas descomponiéndolas en subtareas más sencillas. Al igual que los operadores STN sólo serán aplicables si su precondición se cumple en el estado actual del mundo.

Definición 16 (Método) *Sean N_I, N_X conjuntos disjuntos no vacíos de nombres de individuos y variables respectivamente, y sea $\mathcal{T}$ una TBox acíclica. Un operador $m = (name, task, pre, subtasks)$ es un vector donde:*

- *name, el nombre del método, es una expresión sintáctica de la forma $n(x_1 \dots x_k)$ donde n es un símbolo de operador y $x_1 \dots x_k \in N_X$ son todos los símbolos de variable del operador.*
- *task es la tarea que este método es capaz de descomponer.*
- *$pre(\mathbf{x}, \emptyset)$, la precondición, es una consulta DL que usa el vocabulario de $\mathcal{T}$.*
- *subtasks es una red de subtareas.*

Sea $\sigma : \{x_1 \dots x_k\} \to N_I$ una sustitución que asocia individuos a cada variable. Decimos que el método m es aplicable en el estado K usando la sustitución σ si para todo modelo $\mathcal{I}$ de $\mathcal{K}$ se cumple que $\mathcal{I} \models pre[\sigma]$.

Un método es totalmente ordenado si su red de subtareas es totalmente ordenada.

Durante el proceso de planificación estamos interesados en encontrar métodos y operadores que sean aplicables en el estado actual del sistema y que sean competentes para resolver una tarea aún no resuelta. A continuación formalizamos el concepto de competencia.

Definición 17 (Competencia) *El método* $m = (name, task, pre, subtasks)$ *es competente para resolver la tarea* t *si existe una sustitución* σ *que unifica* $task$ *y* t*. Del mismo modo, el operador* $o = (name, task, pre, post)$ *es competente para resolver la tarea* t *si existe una sustitución* σ *que unifica* $task$ *y* t*.*

La descripción formal del proceso de descomposición de una tarea usando un método es un poco complicada, pero la idea intuitiva que hay detrás es sencilla de entender. Lo que pretendemos es sustituir la tarea resuelta con la red de subtareas del método, y añadir las restricciones de orden necesarias para que las restricciones que antes se aplicaban a la tarea original ahora afecten a todas las nuevas tareas.

Definición 18 (Descomposición de tareas) *Sea* $w = (U, E)$ *una red de tareas,* u *un nodo de* w *sin predecesores, y* m *un método competente para resolver* t_u *usando cierta sustitución* σ*. Sea* $succ(u)$ *el conjunto de sucesores inmediatos de* u*, es decir,* $succ(u) = \{u' \in U \mid (u, u') \in E\}$*. Sea* $succ_1(u)$ *el conjunto de sucesores inmediatos de* u *cuyo único predecesor es* u*. Sea* (U', E') *la red de tareas que resulta de eliminar* u *y todas las aristas que contienen* u*. Sea* (U_m, E_m) *una copia de la red de subtareas de* m*. Si* (U_m, E_m) *no es una red vacía entonces el resultado de descomponer* u *en* w *usando* m *con la sustitución* σ *es el conjunto de redes de tareas:*

$$\delta(w, u, m, \sigma) = \{(U' \cup U_m)[\sigma], E_v[\sigma]) \mid v \in subtasks(m)\}$$

donde

$$E_v = E_m \cup (U_m \times succ(u)) \cup \{(v, u') \mid u' \in succ_1(u)\}$$

En caso contrario $\delta(w, u, m, \sigma) = \{(U'[\sigma], E'[\sigma])\}$

4.2.2. Dominios, problemas y planes

Un dominio de planificación contiene una TBox que describe el vocabulario del dominio y conjuntos de operadores y métodos. Los problemas de planificación completan la descripción del dominio definiendo el estado inicial del problema y la red de tareas objetivo.

Definición 19 (Dominio STN) *Un dominio de planificación STN es un vector* $\mathcal{D} = (\mathcal{T}, \mathcal{M}, \mathcal{O})$ *donde* $\mathcal{T}$ *es una TBox acíclica,* $\mathcal{M}$ *es un conjunto de métodos, y* $\mathcal{O}$ *es un conjunto de operadores.*

Decimos que $\mathcal{D}$ *es un dominio de planificación totalmente ordenado si cada método* $m \in \mathcal{M}$ *es totalmente ordenado.*

Definición 20 (Problema STN) *Un problema de planificación STN* $\mathcal{P} = (\mathcal{T}, \mathcal{M}, \mathcal{O}, \mathcal{A}, w)$ *es un vector donde:*

- $\mathcal{D} = (\mathcal{T}, \mathcal{M}, \mathcal{O})$ *es un dominio de planificación STN.*
- $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *es el estado inicial del problema.*
- w *es la red de tareas objetivo.*

Decimos que $\mathcal{P}$ *es un problema de planificación totalmente ordenado si* w *y* $\mathcal{D}$ *son totalmente ordenados.*

El objetivo del planificador es encontrar un plan solución, es decir una secuencia de acciones que resuelve el problema. Intuitivamente, si un plan π es una solución del problema $\mathcal{P} = (\mathcal{T}, \mathcal{M}, \mathcal{O}, \mathcal{A}, w)$ entonces existe un árbol de descomposición para cada tarea de w cuyas hojas corresponden a las acciones de π. Además, el plan solución debe ser ejecutable y cada método de descomposición del árbol aplicable en el correspondiente estado del sistema. La definición formal de plan solución que mostramos a continuación es recursiva y distingue tres casos.

Definición 21 (Plan) *Un plan* $\pi = \langle \alpha_1 \dots \alpha_n \rangle$ *es una secuencia de acciones. Decimos que el plan* π *es una solución al problema de planificación* $\mathcal{P} = (\mathcal{T}, \mathcal{M}, \mathcal{O}, \mathcal{A}, w)$ *si se cumple alguno de los siguientes casos:*

- ***Caso 1****: la red de tareas* w *es vacía y* π *también es vacío* ($n = 0$).
- ***Caso 2****: existe un nodo* $u \in W$ *sin predecesores,* α_1 *es relevante para la tarea* t_u *y aplicable en* $(\mathcal{T}, \mathcal{A})$, $\gamma((\mathcal{T}, \mathcal{A}), \alpha_1) = (\mathcal{T}, \mathcal{A}')$, *y el plan* $\pi' = (\alpha_2 \dots \alpha_n)$ *es una solución del problema:*

  $$\mathcal{P}' = (\mathcal{T}, \mathcal{M}, \mathcal{O}, \mathcal{A}', w - \{u\})$$

  Intuitivamente, $\mathcal{P}'$ *es el problema de planificación que resulta tras ejecutar la primera acción de* π *y eliminar el nodo de la tarea correspondiente de* w.
- ***Caso 3****: existe un nodo* $u \in w$ *sin predecesores, existe un método* $m \in \mathcal{M}$ *relevante para la tarea* t_u *usando la sustitución* σ *y aplicable en* $(\mathcal{T}, \mathcal{A})$, *y existe un red de tareas* $w' \in \delta(w, u, m, \sigma)$ *tal que* π *es solución de* $\mathcal{P}' = (\mathcal{T}, \mathcal{M}, \mathcal{O}, \mathcal{A}, w')$.

4.2.3. Ejemplo

Al igual que en el ejemplo anterior, usaremos el dominio de la pizzería para describir un problema de planificación jerárquica. El problema que vamos a plantear es totalmente ordenado, es decir, las redes de tareas usan una

relación de orden total. Esto nos permite simplificar la notación y representar las redes de tareas como secuencias.

Los métodos y operadores que vamos a utilizar en este ejemplo se muestran a continuación:

method mPizzaMagherita(?p, ?b, ?t1, ?t2)
task: tPizzaMargherita (?p)
pre: Available(?b), PizzaBase(?b), Available(?t1),
Tomato(?t1), Available(t2), Cheese(?t2)
subtasks: tSelectBase(?p, ?b), tAddTomato(?p, ?t1),
tAddCheese(?p ?t2)

operator oSelectBase(?p, ?b)
task: tSelectBase(?p, ?b)
pre: Available(?b), PizzaBase(?b), NewItem(?p)
post: not NewItem(?p), Pizza(?p), not Available(?b),
hasBase(?p,?b)

operator oAddTomato(?p, ?t)
task: tAddTomato(?p, ?t)
pre: Available(?t), Tomato(?t)
post: not Available(?t), hasTopping(?p,?t)

operator oAddCheese(?p, ?t)
task: tAddCheese(?p, ?t)
pre: Available(?t), Cheese(?t)
post: not Available(?t), hasTopping(?p,?t)

La tarea *tPizzaMargherita* representa la creación de una pizza de tipo margarita, que en la ontología del dominio está definida como una pizza con dos ingredientes de tipo tomate y queso. El método *mPizzaMagherita* resuelve esta tarea descomponiéndola en tres subtareas: *tSelectBase* (elegir una base), *tAddTomato* (añadir tomate a una pizza) y *tAddCheese* (añadir queso a una pizza). Para cada una de estas tareas existe un operador que la resuelve.

El único operador que puede llamarnos la atención es *oSelectBase* que utiliza un individuo de la clase *NewItem* y lo convierte en la nueva pizza. Para evitar la creación de nuevos individuos durante el proceso de planificación hemos definido una clase *NewItem* en la ontología que inicialmente debe contener individuos sin utilizar. No podemos tener inicialmente un individuo de la clase *Pizza* porque la ontología del dominio indica que todas las pizzas tienen base, y en nuestro problema de planificación elegir una base es parte del problema.

El dominio de planificación queda constituido por la ontología de la pizzería y los métodos y operadores anteriores. Para describir un problema de planificación concreto debemos definir el estado inicial y la secuencia de tareas objetivo. El estado inicial será el siguiente:

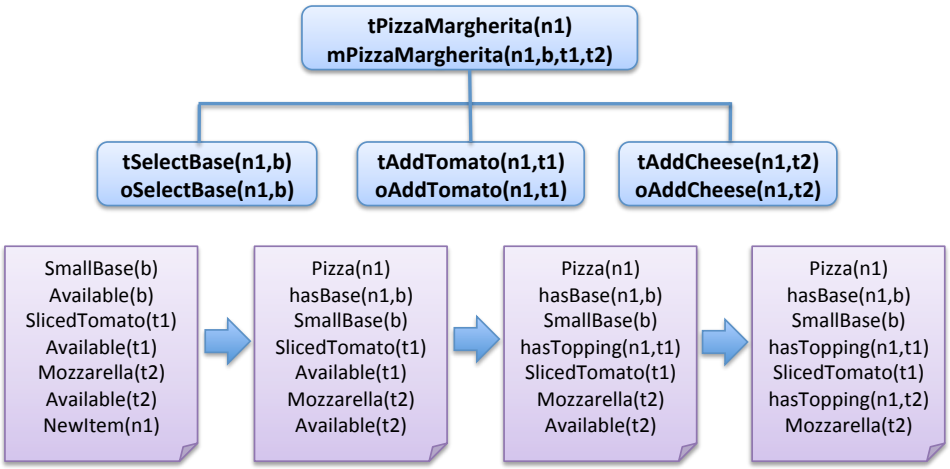


Figura 4.1: Árbol de descomposición

$$SmallBase(b), Available(b), SlicedTomato(t1), Available(t1),$$
$$Mozzarella(t2), Available(t2), NewItem(n1)$$

En este caso disponemos de una base y varios ingredientes. El objetivo de nuestro problema consiste en construir una pizza margarita y lo representamos usando la tarea $tPizzaMargherita(?x)$. El árbol de descomposición que se obtiene tras resolver el problema se muestra en la figura 4.1, y el plan solución es la secuencia de acciones que se obtiene recorriendo las hojas del árbol de izquierda a derecha:

$$\pi = \langle oSelectBase(n1, b), oAddTomato(n1, t1), oAddCheese(n1, t2)\rangle$$

.

4.3. Trabajo relacionado

El uso de DLs para representar dominios y problemas de planificación tiene ventajas evidentes desde el punto de vista de la capacidad expresiva. Sin embargo, los trabajos que tratan el problema de la planificación con DLs son demasiado recientes, y aún es necesario seguir investigando para conseguir que esta tecnología sea realmente efectiva. A continuación resumimos brevemente los trabajos en los que se basan los modelos de planificación descritos en este capítulo y las limitaciones prácticas que aún encontramos en esta tecnología.

En [8] se propone un modelo inicial para integrar formalismos de acciones y DLs muy similar al que nosotros hemos descrito en el apartado 4.1.1. En este trabajo se estudia en profundidad la complejidad computacional de los problemas de proyección y ejecutabilidad usando distintos tipos de DLs. Se concluye que la complejidad de estos problemas coincide con la de comprobar la (in)consistencia de ABoxes en cada una de las DLs extendida con nominales (PSPACE-complete a co-NEXPTIME-complete). También se analizan los problemas semánticos que surgen ante TBoxes con ciclos o si se permite el uso de conceptos arbitrarios en las postcondiciones de las acciones.

En [82] se estudia el problema que surge cuando se utilizan ABoxes para representar el estado de un sistema y operaciones de actualización (*update*) para representar los cambios en el estado. En este trabajo se analizan distintas DLs y se demuestra que es necesario usar la lógica $\mathcal{ALCQIO}^{@}$ para poder expresar los estados actualizados o bien permitir el uso de Boolean ABoxes. El operador "@" se toma de la lógica híbrida y permite construir conceptos de la forma $@_a C$ cuya interpretación es $\Delta^{\mathcal{I}}$ si $a^{\mathcal{I}} \in C^{\mathcal{I}}$ y $\emptyset$ en caso contrario. Las Boolean ABoxes son una generalización de las ABoxes en la que los átomos se relacionan usando conectivas lógicas (por ejemplo disyunciones). En cualquiera de los dos casos lo que conseguimos es la expresividad necesaria para representar las distintas alternativas que surgen como consecuencia de ejecutar una acción a partir de un estado que no conocemos totalmente. En este artículo también se demuestra que aplicando operaciones de actualización se puede producir un crecimiento exponencial de la ABox con respecto a su tamaño original, aunque es posible evitar este problema si permitimos la introducción de conceptos auxiliares en la TBox.

En [92, 91] se extiende el modelo de acciones de [8] y por primera vez se formaliza el problema de la planificación. El trabajo demuestra que la existencia de un plan es decidible si las acciones se describen en subconjuntos de la lógica $\mathcal{ALCQIO}$, y que la complejidad computacional del problema coincide con la del problema de proyección para las lógicas entre $\mathcal{ALC}$ y $\mathcal{ALCQIO}$ si los operadores no contienen postcondiciones condicionales. Para evitar el crecimiento exponencial del estado al aplicar los operadores [82] se usa una representación implícita del estado que almacena los cambios atómicos aplicados al estado inicial del problema.

Además de estos trabajos encontramos otros que exploran otras posibilidades como el uso de TBoxes generales [81] [80] o el uso de otras DLs más ligeras como la familia $\mathcal{EL}$.

Con respecto a la parte de planificación jerárquica el trabajo más representativo utiliza esta tecnología para componer servicios web semánticos [123]. En este trabajo se distinguen dos tipos de servicios web que se representan como operadores STN: los puramente informativos, que sirven para completar el conocimiento que tiene el agente del estado actual, y los que sí tienen efectos que alteran el estado del mundo. En este trabajo se utili-

za la misma semántica para los dos tipos de operadores, lo cual no parece adecuado ya que los primeros corresponden a operaciones de revisión (*revision*) y los segundos a operaciones de actualización (*update*) de bases de conocimiento [74]. Recordemos que una operación de actualización modifica la base de conocimiento cuando el mundo que describe cambia, mientras que una operación de revisión sirve para añadir nuevo conocimiento sobre un mundo estático. Los operadores de planificación que alteran el estado del mundo responden a la categoría de actualizaciones, y sin embargo, la semántica propuesta en [123] está basada en la aproximación WIDTIO (*When in Doubt, Throw it Out*) [146] que es una política de revisión.

4.4. Conclusiones

En este capítulo hemos descrito dos modelos teóricos de planificación basados en DLs que aprovechan la expresividad de estos formalismos para representar tanto el vocabulario del dominio como el estado del sistema. Esta aproximación tiene la ventaja de que permite utilizar ontologías para modelar dominios ricos en conocimiento de manera reutilizable y plantear problemas abstractos en los que el estado inicial no está completamente especificado.

Aunque existen modelos teóricos para este tipo de planificación, aún queda mucho que investigar para conseguir que esta tecnología sea realmente efectiva. Al aumentar la expresividad del lenguaje también aumenta la complejidad de los problemas, por lo que resulta imprescindible disponer de técnicas que permitan representar el espacio de búsqueda y recorrerlo de manera efectiva. Para ello, la comunidad debe estudiar si es posible adaptar los algoritmos que utilizan los planificadores clásicos a dominios en los que el conocimiento se representa usando formalismos expresivos como las DLs. El hecho de que se hayan publicado varios artículos significativos sobre este tema en los últimos años [8, 123, 82, 81, 80, 92, 91], nos hace ser optimistas sobre el desarrollo de esta línea de investigación en el futuro.

Mientras esta tecnología evoluciona, nosotros hemos desarrollado el sistema DLPlan, un planificador basado en DLs que implementa un modelo simplificado en el que las acciones se interpretan de manera sintáctica. En el capítulo siguiente describiremos la arquitectura de este sistema, hablaremos sobre las limitaciones que impone actualizar el estado de forma sintáctica, y mostraremos varios ejemplos de problemas que podemos resolver con este sistema.

Capítulo 5

DLPlan: un planificador basado en OWL-DL

En el capítulo anterior describimos un modelo teórico de planificación que saca partido de la capacidad expresiva de las DLs para representar dominios de planificación con un vocabulario complejo. La idea básica consiste en representar el conocimiento estático usando una base de conocimiento DL donde la parte terminológica representa el vocabulario y los axiomas de dominio, y la parte asertiva se usa para describir el estado actual del sistema. Las acciones, a su vez, se describen de manera declarativa mediante precondiciones y postcondiciones y transforman el estado del sistema modificando los asertos de la ABox. También vimos que los trabajos relacionados con esta línea de investigación son muy recientes y aún queda mucho camino por recorrer para hacer efectiva esta tecnología.

En este capítulo presentamos DLPlan, un planificador de código abierto desarrollado en Java que implementa una simplificación del modelo descrito en el capítulo 4 donde las acciones se interpretan de manera *sintáctica*. DLPlan se integra con Pellet [126], un conocido razonador OWL-DL, y aprovecha las características avanzadas de este sistema para gestionar bases de conocimiento complejas [124, 125]. La arquitectura del sistema trata de separar la gestión del conocimiento, que se delega en Pellet (gestión y actualización del estado, comprobación de la consistencia del modelo, resolución de consultas, ...), del proceso de búsqueda asociado al problema de planificación.

En DLPlan los ficheros que contienen la descripción del dominio de planificación contienen referencias a ontologías que describen el vocabulario y los axiomas del dominio. De este modo intentamos independizar, en la medida de lo posible, el conocimiento general del dominio del conocimiento específico de planificación (operadores, objetivos, ...). Recordemos que las ontologías de dominio pretenden representar el conocimiento asociado a un dominio de manera reutilizable y modular, y por ello deben tratar de modelar el cono-

cimiento de manera neutra, sin tener en cuenta cómo se va a utilizar. En la práctica no es posible independizar totalmente la representación de la información del uso que se le pretende dar, pero debemos tender a ello. Pensemos, por ejemplo, que estas ontologías pueden servir como base de conocimiento para distintos sistemas inteligentes integrados en una arquitectura de negocio que explotan la información del dominio de distintas formas.

Comenzaremos este capítulo describiendo las diferencias entre los modelos teóricos que describimos en el capítulo anterior y la implementación práctica que hemos usado en el sistema DLPlan. La diferencia fundamental es la interpretación sintáctica que se hace de los operadores, que, como veremos, simplifica enormemente la actualización del estado. Más tarde describiremos la arquitectura básica de DLPlan y la sintaxis que se usa para describir los dominios y problemas de planificación. A continuación, mostraremos varios ejemplos de problemas en los que usamos el dominio de las pizzas para mostrar el funcionamiento del sistema. Para finalizar describiremos algunos experimentos en los que comparamos el rendimiento de la versión jerárquica de DLPlan con JSHOP, la versión Java de un conocido planificador HTN [102].

5.1. Diferencias con el modelo teórico

5.1.1. Operadores sintácticos

La principal diferencia que encontramos en DLPlan con respecto al modelo teórico que describimos en el capítulo anterior está en la interpretación sintáctica que se hace de los operadores. En el capítulo anterior cada acción se describe mediante precondiciones y postcondiciones, siendo la postcondición una descripción parcial del estado que se alcanza tras ejecutar la acción. Los cambios concretos que se deben producir en el estado para satisfacer la postcondición dependen de la semántica elegida. Nosotros definimos una semántica en la que las acciones actúan como transformadoras de modelos, y de ese modo nos independizamos de la representación sintáctica de la base de conocimiento. Sin embargo, también vimos que la representación de la base de conocimiento actualizada es problemática por dos motivos [82]: (1) para poder representarla necesitamos aumentar la capacidad expresiva de las lógicas descriptivas usando el constructor "@", y (2) la ABox actualizada puede sufrir un crecimiento exponencial con respecto al tamaño original si no permitimos añadir conceptos auxiliares a la TBox.

La aproximación que usamos en DLPlan es mucho más sencilla, las acciones describen sus efectos mediante listas de axiomas ABox que se deben añadir y eliminar del estado actual. La actualización del estado tras aplicar el operador es *sintáctica* [55, 129] en el sentido de que sólo añadimos y eliminamos axiomas concretos.

Definición 22 (Acción sintáctica) *Sea $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ la base de conocimiento que representa el estado actual. Una acción $\alpha = (pre, del, add)$ es un vector donde pre, del y add con conjuntos finitos de axiomas ABox.*

Decimos que α es aplicable en el estado $\mathcal{K}$ si para todo modelo $\mathcal{I}$ de $\mathcal{K}$ se cumple que $\mathcal{I} \models pre$. El nuevo estado que obtenemos tras su aplicación es $\gamma(\mathcal{K}, \alpha) = (\mathcal{T}, \mathcal{A} \setminus del \cup add)$.

Esta aproximación es muy eficiente si la comparamos con otras aproximaciones, pero debemos tener cuidado porque la actualización sintáctica del estado puede tener consecuencias inesperadas. Por ejemplo, si intentamos eliminar axiomas que aún pueden deducirse a partir del conocimiento restante no obtendremos el efecto deseado. En general escribir operadores de esta forma incrementa la responsabilidad del experto, que debe tener en cuenta las relaciones entre los predicados.

Estos operadores también son dependientes de la representación sintáctica de la base de conocimiento. De hecho, esta aproximación sólo resulta útil en dominios donde conocemos de antemano el tipo de problemas que pretendemos resolver y la manera en la que vamos a representar el estado inicial. Debemos tener en cuenta que los efectos de los operadores se escriben pensando en eliminar y añadir ciertos axiomas que deben aparecer de forma explícita en el estado. Sólo podremos garantizar la validez de los planes generados en los problemas que se ajustan a estas restricciones.

Incluso con todas estas limitaciones, DLPlan resulta útil para resolver cierto tipo de problemas. A lo largo de este capítulo y en el capítulo 7 veremos distintos ejemplos en los que sacamos partido de esta tecnología. Debemos tener en cuenta que, aunque la actualización del estado se hace de manera sintáctica, seguimos aprovechando la capacidad expresiva de las DLs para describir el dominio, detectar inconsistencias en la base de conocimiento, comprobar las precondiciones de los operadores, gestionar eficientemente grandes cantidades de información, etc.

5.1.2. Parámetros de las tareas

La tareas que describimos en el capítulo anterior son átomos cuyos términos sólo pueden ser individuos o variables (que se mapean a individuos de la KB). Para poder definir problemas más interesantes, en DLPlan permitimos, además, el uso de conceptos atómicos, roles y listas. Las listas son secuencias ordenadas de términos y nos permiten describir tareas con un número indeterminado de términos. Por ejemplo, la expresión *[x1 x2 (. ?xs)]* representa una lista donde los dos primeros elementos son *x1* y *x2*, y la variable *?xs* representa el resto de la lista, que puede o no ser vacía.

A continuación mostramos una tarea que representa la creación de una pizza pequeña con base fina y dos ingredientes vegetarianos. Como usamos una lista podemos reutilizar la misma tarea para representar la creación de

otras pizzas con más o menos ingredientes. En este caso el primer término de la tarea es un concepto de la ontología que indica el tamaño de la pizza, el segundo término otro concepto que indica el tipo de base, el tercer término es la lista con los tipos de ingredientes, y el último término es una variable que se asociará al individuo que represente la pizza cuando la hayamos creado.

tMakePizzaByTypes(Small, ThinBase, [VegetarianTopping, VegetarianTopping], ?pizza)

5.1.3. Otras consideraciones

DLPlan admite dominios de planificación jerárquicos y no jerárquicos. Los dominios no jerárquicos (a veces nos referiremos a ellos como dominios clásicos) se describen por medio de las ontologías del dominio y los operadores de planificación. El objetivo en estos problemas es encontrar secuencias de acciones que llevan a un estado donde se cumple el objetivo, que se representa mediante una consulta conjuntiva DL de tipo booleano. La implementación actual de DLPlan trata de resolver el problema usando una búsqueda ciega en el espacio de estados (con un algoritmo de búsqueda en profundidad iterativa). Como no usamos heurísticas, sólo podremos resolver, en un tiempo razonable, problemas en los que la solución consta de unas pocas acciones.

El espacio de estados es en realidad un espacio de *estados abstractos*, en el sentido de que el conocimiento se interpreta usando la hipótesis de mundo abierto y por tanto cada estado puede contener varios (o incluso infinitos) modelos. Eso quiere decir que en cada estado abstracto representamos de manera compacta un conjunto (posiblemente infinito) de estados posibles.

Los dominios jerárquicos se describen usando ontologías del dominio, tareas, métodos y operadores. La implementación actual de DLPlan usa un modelo STN totalmente ordenado por lo que las redes de tareas se describen usando secuencias. Igual que en JSHOP, los métodos y operadores se intentan aplicar en el mismo orden en el que están definidos en el fichero que describe el dominio de planificación. Usando la aproximación jerárquica podemos calcular planes con muchas más acciones, ya que las tareas y métodos, si están bien escritos, restringen considerablemente el espacio de búsqueda.

En el modelo teórico del capítulo anterior se usa la idea intuitiva de que el vocabulario del dominio se describe usando la parte terminológica de la base de conocimiento, y que la parte asertiva se reserva para representar el estado. Sin embargo, muchas ontologías también utilizan individuos para representar el vocabulario del dominio. Estos individuos representan entidades comunes a cualquier estado del mundo. En realidad esto no supone ningún problema práctico, en DLPlan permitimos que las ontologías del dominio puedan contener individuos y usamos distintos espacios de nombre para separar los individuos del dominio y del problema.

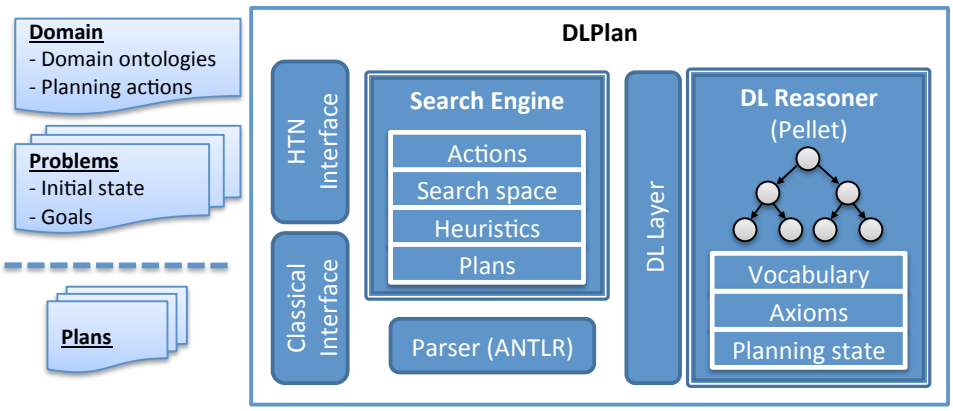


Figura 5.1: Arquitectura de DLPlan

Finalmente, debemos mencionar que DLPlan utiliza de manera intensiva la capacidad de razonamiento incremental de Pellet [56]. Cada vez que aplicamos una acción añadimos y eliminamos axiomas de la ABox. Resulta imprescindible, por tanto, disponer de un razonador capaz de gestionar de manera eficiente estos pequeños cambios, es decir, que actualice sólo la parte de la base de conocimiento que se ve afectada. El estudio de estas técnicas de razonamiento incremental también es bastante reciente [107, 54]. En Pellet podemos usar esta característica si la ontología se describe usando las lógicas $\mathcal{SHOQ}$ o $\mathcal{SHIQ}$, que representan grandes subconjuntos de OWL-DL.

5.2. Arquitectura del sistema

La arquitectura básica de DLPlan se muestra en la figura 5.1. Este diseño trata de separar la parte relativa a la gestión del conocimiento estático, que se delega en un razonador DL, de la parte que representa y guía el proceso de búsqueda. Para independizar al sistema del razonador concreto que estemos usando, la comunicación entre ambos módulos se realiza a través de una sencilla capa de abstracción que en la figura aparece con el nombre *DL Layer*. El razonador es responsable de cargar las ontologías del dominio, comprobar la consistencia de la base de conocimiento, verificar si los operadores son aplicables en el estado actual, actualizar el estado añadiendo y eliminando asertos, etc. El motor de búsqueda, por su parte, es el responsable de resolver el problema de planificación y para ello debe decidir cómo representar y recorrer el espacio de búsqueda.

El planificador recibe como entrada dos ficheros que contienen la descripción del dominio y la descripción del problema de planificación. Dependiendo de si estamos o no trabajando con un dominio de planificación jerárquico, el contenido de estos ficheros será diferente, pero en cualquier caso el fichero

de dominio debe contener las URIs de las ontologías OWL-DL que describen el vocabulario y las restricciones del dominio. Estas ontologías pueden contener referencias a otras ontologías de manera recursiva, lo que nos permite modelar dominios complejos de manera modular. El fichero de descripción del problema contiene los axiomas ABox que describen el estado inicial y los objetivos del problema.

A la hora de implementar la capa de abstracción que independiza el motor de búsqueda del razonador DL que estamos usando barajamos varias opciones:

- *ATERMS* (*Annotated Terms*) [24]. Esta es la biblioteca que utiliza internamente Pellet para representar el conocimiento, y tiene la ventaja de que optimiza el espacio necesario para representar la información maximizando la compartición de subtérminos. Esta tecnología permite gestionar el conocimiento de manera eficiente, pero resulta muy poco intuitiva porque no representa explícitamente los elementos que esperamos encontrar en una ontología (clases, roles, individuos, axiomas, ...). Los mismos autores de Pellet recomiendan acceder al razonador con una interfaz de más alto nivel.

- *Jena* [1]. Este armazón Java para construir aplicaciones de la web semántica ha conseguido bastante popularidad. Proporciona un entorno común para gestionar información descrita en RDF, RDFS y OWL, gestiona la persistencia de la información, y proporciona un motor para responder consultas SPARQL. Al intentar abarcar tantas tecnologías distintas es un armazón bastante pesado, por lo que realizar conversiones entre la representación interna de Pellet y las clases de Jena consume bastante tiempo. Además, la visión que se proporciona de las ontologías OWL-DL no es la que un experto en lógica esperaría, ya que está ligada al resto de tecnologías que se utilizan para la persistencia (RDFS y RDF).

- *OWL-API* [66]. Esta biblioteca define una capa de abstracción ligera que permite trabajar con ontologías OWL-DL desde el punto de vista lógico, independizando al usuario del resto de tecnologías y definiendo una interfaz mucho más sencilla que la de Jena. Desafortunadamente, OWL-API aún no contiene clases para realizar consultas DL sobre una base de conocimiento, y nosotros necesitamos esa funcionalidad para comprobar si se cumplen las precondiciones de los operadores.

Todas estas opciones tienen ventajas e inconvenientes. OWL-API parece la opción más cercana a lo que nosotros necesitamos pero no es fácil extender esta biblioteca para añadir la parte de consultas. Al final decidimos crear nuestra propia capa de abstracción con un interfaz mínimo que sólo contiene lo imprescindible para el planificador. La traducción entre los tipos de la

interfaz y la representación interna usada por Pellet es muy sencilla y rápida, y creemos que sería igualmente sencillo adaptarla a otro razonador similar.

5.2.1. Ciclo de funcionamiento

Las ontologías OWL-DL que usamos para modelar los dominios pueden llegar a ser bastante complejas y contener mucha información. El diseño modular de las ontologías promueve que el conocimiento se reutilice, por lo que resulta común encontrar ontologías cuyos axiomas dependen de entidades definidas en otras ontologías que, a su vez, pueden referenciar a más ontologías. Recordemos que la ontología de las pizzas, un ejemplo muy sencillo que se utiliza para enseñar a modelar conocimiento con OWL-DL, contiene ya 100 clases y varios cientos de axiomas.

Después de cargar la ontología del dominio (o las ontologías si son varias) en Pellet, necesitamos realizar una *clasificación* inicial de la base de conocimiento antes de empezar a operar sobre ella. Durante la clasificación se calculan las relaciones de subsunción entre todos los conceptos atómicos, y se crean las estructuras necesarias para acelerar los subsiguientes procesos de inferencia. Esta es una operación costosa que puede llevar unos pocos segundos o tardar varias horas dependiendo de la complejidad y tamaño de las ontologías. Afortunadamente, este proceso sólo es necesario realizarlo una vez. Una vez hayamos clasificado la ontología del dominio podemos empezar a resolver todos los problemas que se definan en dicho dominio.

El ciclo de funcionamiento de DLPlan se puede resumir en los siguientes pasos:

1. Leer el fichero que contiene la descripción del dominio de planificación.
2. Cargar las ontologías del dominio en el razonador.
3. Clasificar la base de conocimiento.
4. Resolver cada uno de los problemas:
   a) Leer el fichero que contiene la descripción del problema.
   b) Cargar el estado inicial en la base de conocimiento y comprobar su consistencia.
   c) Recorrido del espacio de búsqueda hasta alcanzar una solución.
   d) Eliminar de la base de conocimiento la información sobre el estado.

La parte relativa a recorrer el espacio de búsqueda depende de si estamos resolviendo problemas en dominios clásicos o jerárquicos, y la trataremos más adelante.

5.3. Descripción de dominios y problemas

En los siguiente apartados introduciremos la sintaxis concreta que se usa en DLPlan para describir los dominios y problemas de planificación. También comentaremos algunas peculiaridades que surgen como consecuencia de usar ontologías OWL-DL para describir el vocabulario del dominio e interpretar el conocimiento usando la hipótesis de mundo abierto.

5.3.1. Ontologías y espacios de nombres

Ya hemos comentado que los ficheros que describen el dominio de planificación deben indicar cuáles son las ontologías en las que se describe el vocabulario y las restricciones del dominio. Las ontologías son recursos de la red y como tales se identifican mediante URIs, largas cadenas de texto que permiten acceder a la información usando protocolos estándar. En DLPlan, las ontologías del dominio se representan usando la sintaxis siguiente:

ontology: <http:*//gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl*>

Sin embargo, a veces resulta cómodo trabajar con copias locales de las ontologías y de ese modo evitar conexiones a Internet durante el proceso de búsqueda. Podemos indicar a DLPlan que debe cargar una ontología desde un fichero local, y no desde la dirección de Internet, añadiendo la ruta del archivo después de la URI:

ontology: <http:*//gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl*>,
pizzeria .owl;

Las entidades de una ontología también se identifican mediante URIs, por ejemplo el concepto *Pizza* de la ontología anterior se identifica mediante la URI `http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl#Pizza`. Este tipo de cadenas son tediosas de escribir, por lo que se suele recurrir a prefijos. Por ejemplo, usando la siguiente declaración de prefijos en DLPlan podemos referirnos al concepto *Pizza* usando la cadena *dom:Pizza*.

prefix dom: <http:*//gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl#*>

En el resto del capítulo usaremos el prefijo *dom* para referirnos a elementos del dominio y el prefijo *prob* para referirnos a elementos del problema.

5.3.2. Operadores

Los operadores clásicos tienen la forma $o = (name, pre, del, add)$ donde *pre* es una consulta DL y *del* y *add* son listas de átomos que pueden contener variables distinguibles de la precondición. Para saber si un operador

es aplicable en el estado actual basta con usar su precondición como consulta sobre la base de conocimiento que representa el estado actual. Si la consulta no tiene respuestas significa que el operador no es aplicable. Si obtenemos respuestas, cada una de ellas será una sustitución σ que asociará las variables distinguibles de la precondición a individuos del estado actual. Aplicando dicha sustitución a las listas *del* y *add* obtenemos los axiomas ABox que debemos eliminar y añadir del estado para ejecutar la acción correspondiente. Como sabemos, los axiomas que hemos añadido y eliminado cada vez que aplicamos una acción, es sencillo hacer *backtracking* durante la búsqueda cuando sea necesario.

Otro detalle interesante es que, debido a la interpretación de mundo abierto, eliminar axiomas de la ABox siempre resulta en una pérdida de conocimiento. Lo que realmente estamos haciendo al eliminar asertos es aumentar los posibles modelos de la base de conocimiento.

A continuación mostramos un ejemplo de operador usando la sintaxis de DLPlan:

oper addTopping
vars: ?p, ?t
pre: dom:NonCookedPizza(?p), dom:AvailableTopping(?t)
del: dom:AvailableTopping(?t)
add: dom:hasTopping(?p,?t)

El operador *addTopping* añade un ingrediente a una pizza. La precondición establece que sólo podemos realizar esta acción si el ingrediente está disponible y la pizza aún no ha sido cocinada. Para ver si este operador es aplicable en el estado actual efectuamos la siguiente consulta:

$$Q(\{?p, ?t\}, \emptyset) = \{NonCookedPizza(?p), AvailableTopping(?t)\}$$

Supongamos que en el estado actual existe un ingrediente disponible *t1* y una pizza sin cocinar *p1*. Al efectuar la consulta anterior obtenemos la sustitución $\sigma = \{?p/p1, ?t/t1\}$ por lo que la acción correspondiente eliminará el axioma *AvailableTopping(t1)* del estado y añadirá *hasTopping(p1,t1)*. En realidad, al eliminar el axioma *AvailableTopping(t1)* no estamos diciendo que *t1* ya no esté disponible, sino que no sabemos si lo está. Esto no es problemático porque los operadores de nuestro dominio siempre comprueban si un ingrediente esta disponible antes de usarlo, así que *t1* no se volverá a considerar.

Otra consecuencia del mundo abierto es que no podemos saber el número de ingredientes de una pizza a no ser que lo indiquemos explícitamente. Usando el operador *addTopping* vamos añadiendo asertos *hasTopping* al estado por lo que podemos saber el número mínimo de ingredientes de la pizza, pero no el número real (puede tener más). Si queremos llevar la cuenta, necesitamos escribir operadores específicos como el que mostramos a continuación.

oper addTopping0
vars: ?p, ?t
pre: dom:NonCookedPizza(?p), dom:AvailableTopping(?t),
dom:PizzaWith0Toppings(?p)
del: dom:AvailableTopping(?t), dom:PizzaWith0Toppings(?p)
add: dom:hasTopping(?p, ?t), dom:PizzaWith1Toppings(?p)

5.3.3. Operadores y métodos STN

En planificación STN, además de comprobar cuándo un operador o método es aplicable en el estado actual, también debemos determinar si es competente a la hora de resolver una tarea. En DLPlan usamos planificación STN totalmente ordenada, por lo que la red de tareas objetivos es siempre una secuencia y podemos ir resolviendo las tareas en orden. El proceso es el siguiente:

1. Se elige la primera tarea t de la secuencia de tareas objetivo.
2. Se buscan los métodos y operadores competentes para resolver t.
   a) Si un método / operador es competente para resolver la tarea t entonces existe una sustitución σ que unifica t y la tarea del método / operador.
   b) Aplicamos la sustitución σ al método / operador. Llegados a este punto todas las variables que aparecen en la precondición del método / operador deben representar individuos.
3. Se comprueba cuáles de los métodos / operadores competentes son aplicables en el estado actual usando sus precondiciones como consultas DL.

A continuación mostramos un operador STN usando la sintaxis concreta de DLPlan. En este caso el operador resuelve la tarea *tAddTopping* (que representa la adición de un ingrediente a una pizza) cuando la pizza no tiene ingredientes:

oper oAddTopping0
task: tAddTopping(?p, ?t)
vars:
pre: dom:PizzaWith0Toppings(?p), dom:NonCookedPizza(?p),
dom:AvailableTopping(?t)
del: dom:AvailableTopping(?t), dom:PizzaWith0Toppings(?p)
add: dom:hasTopping(?p ?t), dom:PizzaWith1Toppings(?p)

El siguiente método descompone la tarea *tMakePizzaByTypes*, que se usa para crear una pizza con ciertas características e ingredientes, en 3 subtareas: seleccionar la base adecuada, añadir los ingredientes y hornear la pizza.

method mMakePizzaByTypes
task: tMakePizzaByTypes(?size, ?baseType, ?toppingTypes, ?pizza)
vars:
pre:
subtasks: tSelectBase(?size, ?baseType, ?pizza),
tAddToppingsByType(?pizza, ?toppingTypes), tBakePizza(?pizza)

En este caso las variables *?size* y *?basetype* representan conceptos de la ontología, *?toppingTypes* es una lista y *?pizza* representa al individuo que describe la pizza resultante.

5.3.4. Estado inicial

El estado inicial se define mediante axiomas ABox que usan el vocabulario del dominio. Estos axiomas se pueden leer de una ontología o podemos representarlos en el fichero de descripción del problema. Por ejemplo, el siguiente fragmento describe un estado inicial donde hay una base fina y 3 ingredientes distintos:

dom:Available(prob:b1), dom:ThinBase(prob:b1),
dom:Available(prob:t1), dom:Fish(prob:t1),
dom:Available(prob:t2), dom:Pepper(prob:t2),
dom:Available(prob:t3), dom:SpicyTopping(prob:t3)

En OWL-DL no se asume que dos individuos son distintos sólo porque tienen nombres diferentes. En el fichero de descripción del problema podemos indicar qué individuos son distintos usando el predicado *Different(i1, . . . , iN)*, que se usa en DLPlan para indicar que los individuos *i1*, . . . , *iN* son distintos dos a dos. También podemos indicar que todos los individuos del estado son distintos usando la palabra reservada *AllDifferent*.

5.3.5. Objetivos

Los objetivos en planificación clásica se representan usando una consulta conjuntiva DL. Por ejemplo, en DLPlan el siguiente código define el objetivo de crear una pizza con 3 ingredientes donde al menos uno de ellos es un tipo de carne:

goals: dom:PizzaWith3Toppings(?x), dom:hasTopping(?x,?y),
dom:Meat(?y)

En la versión STN los objetivos se representan como una secuencia de tareas que se deben resolver. A continuación mostramos el objetivo de crear una pizza familiar con base fina y 2 ingredientes, donde al menos uno de ellos es picante.

goals: tMakePizzaByTypes(dom:Familiar, dom:ThinBase,
[dom:PizzaTopping, dom:SpicyTopping], ?pizza)

Las variables que aparecen en las descripciones de los objetivos sólo pueden representar individuos, nunca conceptos o roles. En realidad esto no supone una limitación ya que siempre podemos usar conceptos o roles más abstractos si necesitamos representar objetivos más generales. Por ejemplo, si en el objetivo anterior no nos importa el tamaño de la pizza ni el tipo de base podemos usar los conceptos *PizzaBase* y *Size* como parámetros de la tarea.

5.3.6. Preprocesador

A veces los dominios de planificación contienen operadores que son muy similares entre sí. En el dominio de la pizzería, por ejemplo, si queremos controlar el número de ingrediente de una pizza según la vamos construyendo, necesitamos definir varios operadores (*addTopping0*, *addTopping1*, ...).

DLPlan incorpora un modesto preprocesador que permite escribir todos estos operadores de manera compacta. Usando unas variables de preprocesador, que se definen al principio del fichero y toman valores dentro de un rango, y sencillas expresiones aritméticas podemos escribir todos los operadores *addToppingX* de una vez:

```
|N=0..5|

oper addTopping|N|
vars: ?p, ?t
pre: dom:NonCookedPizza(?p), dom:AvailableTopping(?t),
     dom:PizzaWith|N|Toppings(?p)
del: dom:AvailableTopping(?t), dom:PizzaWith|N|Toppings(?p)
add: dom:hasTopping(?p, ?t), dom:PizzaWith|N+1|Toppings(?p)
```

5.4. Ejemplos

Una vez hemos descrito las distintas piezas que necesitamos para definir dominios y problemas de planificación, vamos a mostrar algunos ejemplos concretos basados en nuestro ya conocido dominio de las pizzas. Los dos dominios de planificación que mostraremos usan la misma ontología del dominio, pero uno es un dominio de planificación clásico y otro jerárquico.

5.4.1. Planificación clásica

La figura 5.2 muestra la descripción del dominio que vamos a utilizar en este ejemplo. La mayor parte de este código ya lo hemos explicado en las secciones anteriores, así que sólo lo repasaremos de forma breve.

```
|N=0..5|
/* Variable de preprocesador para escribir los operadores addToppingX. */

name: domain1  /* Nombre del dominio. */
type: classical  /* Dominio de tipo no jerárquico */

/* URI de la ontología que define el vocabulario y los axiomas del dominio */
ontology: <http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl>

/* Prefijo para referirnos a las entidades de la ontología. */
prefix dom: <http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl#>

/* Elige una base y crea una nueva pizza sin ingredientes. */
oper selectBase
vars: ?pizza, ?base
pre: dom:NewItem(?pizza), dom:AvailableBase(?base)
del: dom:NewItem(?pizza), dom:AvailableBase(?base)
add: dom:AvailablePizza(?pizza), dom:NonCookedPizza(?pizza),
     dom:hasBase(?pizza, ?base), dom:PizzaWith0Toppings(?pizza)

/* Añade un ingrediente a una pizza. El preprocesador creará 6 operadores
 * concretos (addTopping0, ..., addTopping5). */
oper addTopping|N|
vars: ?pizza, ?topping
pre: dom:NonCookedPizza(?pizza), dom:AvailableTopping(?topping),
     dom:PizzaWith|N|Toppings(?pizza)
del: dom:AvailableTopping(?topping), dom:PizzaWith|N|Toppings(?pizza)
add: dom:hasTopping(?pizza, ?topping), dom:PizzaWith|N+1|Toppings(?pizza)

/* Cocina una pizza. */
oper bakePizza
vars: ?pizza
pre: dom:NonCookedPizza(?pizza)
del: dom:NonCookedPizza(?pizza)
add: dom:CookedPizza(?pizza)
```

Figura 5.2: Ejemplo de dominio clásico

La descripción del dominio comienza definiendo la variable del preprocesador N que usaremos después para generar automáticamente los operadores que añaden ingredientes a una pizza. A continuación indicamos el nombre y tipo del dominio que estamos definiendo.

Después indicamos la URI de la ontología que contiene el vocabulario y el prefijo que usaremos para referirnos a sus entidades usando una sintaxis sencilla. Si lo necesitamos podemos añadir varias secciones *ontology* y *dom* para referenciar más ontologías.

La descripción del dominio continua con la descripción de los operadores de planificación, que en este caso nos servirán para crear distintos tipos de pizzas combinando ingredientes.

El operador *selectBase* elige una base y crea una nueva pizza. Para crear una pizza necesitamos un nuevo individuo en la KB que la represente pero, para mantener las cosas sencillas, no queremos crear nuevos individuos dinámicamente. Lo que hacemos es definir un concepto *NewItem* en la ontología que inicialmente contiene individuos *sin usar*. El operador *selectBase* toma uno de esos individuos y una base y crea una nueva pizza sin ingredientes.

La siguiente definición (*addTopping*$|N|$) será expandida por el preprocesar para crear los correspondientes operadores que añaden un ingrediente a una pizza en función del número de ingredientes que ya tenga la pizza. Por último, el operador (*bakePizza*) cocina una pizza a la que ya no se podrán añadir nuevos ingredientes.

La figura 5.3 muestra la descripción de un problema de planificación asociado al dominio anterior. El fichero comienza dando un nombre al problema, indicando el dominio en el que se enmarca, y definiendo los prefijos que simplificarán la escritura del resto del problema. A continuación se definen los axiomas ABox que representan el estado inicial del problema, en este caso disponemos de dos tipos de base y varios ingredientes para crear pizzas. También sabemos que todos los individuos representan entidades diferentes al incluir la palabra *AllDifferent*. El objetivo que perseguimos en este problema consiste en crear una pizza vegetariana y picante con 3 ingredientes.

Este problema admite varias soluciones, a continuación mostramos una de ellas:

selectBase(prob:b2, prob:item1), addTopping0(prob:t3, prob:item1),
addTopping1(prob:t2, prob:item1), addTopping2(prob:t5, prob:item1)

5.4.2. Planificación STN

Usando planificación jerárquica podemos resolver problemas de manera mucho más eficiente, ya que la descripción del dominio contiene información explícita que permite guiar la búsqueda. La definición del dominio STN que usaremos en este ejemplo se muestra en las figuras 5.4 y 5.5. La parte inicial

```
name: problem2    /* Nombre del problema */
type: classical   /* Tipo de problema */
domainName: domain1 /* Dominio de planificación asociado */

/* Prefijos para referirnos a las entidades. */
prefix dom: <http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl#>
prefix prob: <http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/prob2.owl#>

/* Descripción del estado inicial */
initState:
dom:AvailableBase(prob:b1),    dom:SmallBase(prob:b1),
dom:AvailableBase(prob:b2),    dom:ThinBase(prob:b2),
dom:AvailableTopping(prob:t1), dom:Fish(prob:t1),
dom:AvailableTopping(prob:t2), dom:Pepper(prob:t2),
dom:AvailableTopping(prob:t3), dom:Mozzarella(prob:t3),
dom:AvailableTopping(prob:t4), dom:Tomato(prob:t4),
dom:AvailableTopping(prob:t5), dom:SpicyTopping(prob:t5),
dom:VegetarianTopping(prob:t5), dom:NewItem(prob:item1),
AllDifferent

/* Objetivo: pizza vegetariana y picante con 3 ingredientes. */
goals:
dom:VegetarianPizza(?x), dom:SpicyPizza(?x), dom:PizzaWith3Toppings(?x)
```

Figura 5.3: Ejemplo de problema clásico

del fichero de dominio es muy similar a la que ya vimos en el ejemplo anterior, sólo que ahora usamos la palabra reservada *stn* en lugar de *classical* para indicar que se trata de un dominio jerárquico.

A continuación se definen los operadores STN equivalentes a los operadores clásicos que vimos en el apartado anterior. Las tareas *tSelectBase*, *tAddTopping* y *tBakePizza* representan, respectivamente, la creación de una nueva pizza usando una base de cierto tipo y tamaño, añadir un ingrediente a una pizza, y hornear la pizza. Para cada una de estas tareas existe un operador que la resuelve.

El dominio continua definiendo dos formas de resolver la tarea *tAddToppingsByType*, que representa la adición de ingredientes a una pizza a partir de sus tipos. Si la lista de tipos de ingredientes no es vacía, la tarea se resolverá por descomposición usando el método *mAddToppingsByType*. Este método elige un ingrediente del primer tipo y descompone la tarea inicial en dos tareas más sencillas: añadir el ingrediente seleccionado y añadir el resto de ingredientes de manera recursiva. Cuando la lista de ingredientes sea vacía podremos aplicar el operador *oDoNothing* acabando con la recursión.

Finalmente, la tarea *tMakePizzaByTypes* representa el objetivo de construir una pizza dados su tamaño, tipo de base, y tipos de ingredientes. El

```
|N=0..5|
/* Variable de preprocesador para escribir los operadores addToppingX. */

name: domain1  /* Nombre del dominio. */
type: stn      /* Dominio de tipo jerárquico */

/* URI de la ontología que define el vocabulario y los axiomas del dominio */
ontology: <http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl>

/* Prefijo para referirnos a las entidades de la ontología. */
prefix dom: <http://gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl#>

/* Selecciona una base a partir de su tamaño y tipo, y crea una nueva
 * pizza sin ingredientes. */
oper oSelectBase
task: tSelectBase(?size, ?baseType, ?pizza)
vars: ?base
pre: dom:NewItem(?pizza), dom:AvailableBase(?base), ?size(?base),
     ?baseType(?base)
del: dom:NewItem(?pizza), dom:AvailableBase(?base)
add: dom:AvailablePizza(?pizza), dom:NonCookedPizza(?pizza),
     dom:hasBase(?pizza, ?base), dom:PizzaWith0Toppings(?pizza)

/* Añade un ingrediente a una pizza. El preprocesador creará 6 operadores
 * concretos (addTopping0, ..., addTopping5). */
oper oAddTopping|N|
task: tAddTopping(?pizza, ?topping)
vars:
pre: dom:PizzaWith|N|Toppings(?pizza), dom:NonCookedPizza(?pizza),
     dom:AvailableTopping(?topping)
del: dom:AvailableTopping(?topping), dom:PizzaWith|N|Toppings(?pizza)
add: dom:hasTopping(?pizza, ?topping), dom:PizzaWith|N+1|Toppings(?pizza)

/* Cocina una pizza. */
oper oBakePizza
task: tBakePizza(?pizza)
vars:
pre: dom:NonCookedPizza(?pizza)
del: dom:NonCookedPizza(?pizza)
add: dom:CookedPizza(?pizza)
```

Figura 5.4: Ejemplo de dominio STN (1/2)

método *mMakePizzaByTypes* resuelve esta tarea descomponiéndola en 3 subtareas más sencilla: seleccionar la base, añadir los ingredientes y cocinar la pizza.

La figura 5.6 muestra un ejemplo de problema definido en el contexto del

```
/* Operador que no hace nada. Es necesario para acabar la recursión. */
oper oDoNothing
task: tAddToppingsByType(?pizza, [])
vars:
pre:
del:
add:

/* Añade ingredientes a una pizza dados sus tipos de forma recursiva. */
method mAddToppingsByType
task: tAddToppingsByType(?pizza, [?toppingtype . ?l])
vars: ?topping
pre: dom:NonCookedPizza(?pizza), dom:AvailableTopping(?topping),
     ?toppingtype(?topping)
subtasks: tAddTopping(?pizza, ?topping), tAddToppingsByType(?pizza, ?l)

/* Crea una pizza dados su tamaño, tipo de base y tipos de ingredientes. */
method mMakePizzaByTypes
task: tMakePizzaByTypes(?size, ?baseType, ?toppingTypes, ?pizza)
vars:
pre:
subtasks: tSelectBase(?size, ?baseType, ?pizza),
          tAddToppingsByType(?pizza, ?toppingTypes),
          tBakePizza(?pizza)
```

Figura 5.5: Ejemplo de dominio STN (2/2)

dominio anterior. El objetivo en este caso es construir una pizza pequeña con masa gruesa y dos ingredientes de tipo vegetal. La figura 5.7 muestra el árbol de descomposición asociado a una de las posibles soluciones del problema.

5.5. Tipos de problemas que podemos resolver

En la introducción del capítulo ya dijimos que al usar operadores sintácticos, en general, no podemos garantizar la validez de los planes generados. Sin embargo, cuando los problemas que se plantean se ajustan al tipo de problemas para los que se diseñó el dominio, sí es posible garantizar su validez. Por ejemplo, las dos versiones anteriores del dominio de las pizzas (jerárquica y no jerárquica) se diseñaron para resolver problemas en los que se proporcionan un conjunto de bases e ingredientes y se pretende construir determinados tipos de pizza. Dentro de este tipo de problemas sí vamos a gestionar correctamente cierto indeterminismo en el estado inicial, lo que nos permite construir planes cuando la información que disponemos sobre las bases e ingredientes es incompleta.

Las pizzas que construyamos estarán relacionadas con su base mediante el

name: problem1 */* Nombre del problema */*
type: stn */* Tipo de problema */*
domainName: domain1 */* Dominio de planificación asociado */*

/ Prefijos para referirnos a las entidades. */*
prefix dom: <http://*gaia.fdi.ucm.es/dlplan/pizzeria/1.0/dom.owl#*>
prefix prob: <http://*gaia.fdi.ucm.es/dlplan/pizzeria/1.0/prob2.owl#*>

/ Descripción del estado inicial */*
initState:
dom:AvailableBase(prob:b1), dom:DeepPanBase(prob:b1),
dom:SmallBase(prob:b1), dom:AvailableBase(prob:b2),
dom:RegularBase(prob:b2), dom:DeepPanBase(prob:b2),
dom:AvailableTopping(prob:t1), dom:Fruit(prob:t1),
dom:AvailableTopping(prob:t2), dom:Olive(prob:t2),
dom:AvailableTopping(prob:t3), dom:Mozzarella(prob:t3),
dom:AvailableTopping(prob:t5), dom:Meat(prob:t5),
dom:AvailableTopping(prob:t6), dom:SpicyTopping(prob:t6),
dom:AvailableTopping(prob:t7), dom:Vegetable(prob:t7),
dom:SpicyTopping(prob:t7), dom:NewItem(prob:item1),
AllDifferent

/ Objetivo: pizza mediana con masa gruesa y 2 ingredientes de tipo vegetal */*
goals: tMakePizzaByTypes(dom:Small, dom:DeepPanBase,
[dom:Vegetable, dom:Vegetable], ?pizza)

Figura 5.6: Ejemplo de problema STN

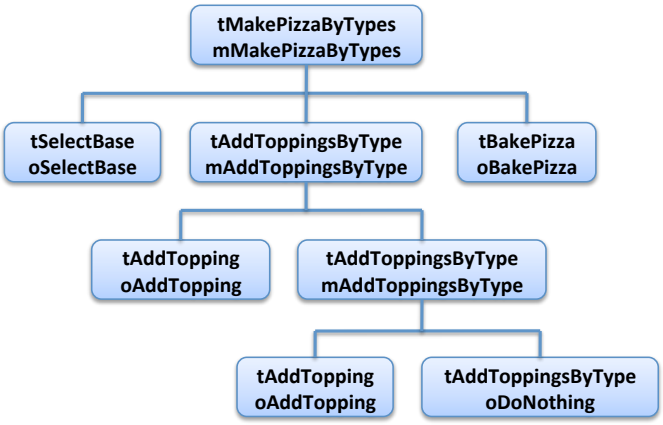


Figura 5.7: Ejemplo de árbol de descomposición

role *hasBase* y con sus ingredientes con *hasTopping*, por lo que podemos utilizar las capacidades de inferencia de las DLs para clasificar los ingredientes y pizzas debajo de los correspondientes conceptos definidos de la ontología. En los ejemplos anteriores ya hemos visto cómo es posible usar conceptos de la parte media o superior de la ontología para representar de manera natural descripciones abstractas del estado inicial. Por ejemplo, en el problema no jerárquico definíamos un ingrediente *t5* de tipo picante y vegetariano pero no dábamos más información sobre él. Tampoco necesitamos más información para resolver el problema y, de hecho, usamos ese ingrediente abstracto en la solución del problema.

Vamos a plantear un último problema STN en el que mostraremos cómo se gestiona el conocimiento incompleto del estado inicial. Nuestro objetivo en este caso consiste en construir una pizza pequeña con dos ingredientes, uno de tipo vegetariano y otro de tipo picante:

```
tMakePizza(dom:Small, dom:PizzaBase,
           [dom:VegetarianTopping, dom:SpicyTopping], ?p)
```

En esta ocasión usaremos un estado inicial mucho más abstracto: *b1* es una base de pizza pequeña, *t1* es un tipo de queso o un vegetal, y *t2* es un ingrediente picante pero no es pimienta. Para representar este estado inicial en DLPlan necesitamos introducir en la ontología del dominio los correspondientes conceptos, pero el estado que representaríamos sería equivalente al siguiente:

$$((Small \sqcap PizzaBase \sqcap Available)\ b1)$$
$$((Cheese \sqcup Vegetable) \sqcap Available\ t1)$$
$$((SpicyTopping \sqcap \neg Pepper) \sqcap Available\ t2)$$
$$((NewItem\ p1)$$

La figura 5.8 resume el proceso que seguiría DLPlan para resolver este problema. En el lado izquierdo de la figura mostramos la evolución del árbol de descomposición de tareas, y en el lado derecho la correspondiente evolución del estado. Durante el proceso de planificación se hacen ciertas inferencias interesantes, por ejemplo *t1* se elije como ingrediente vegetariano porque el razonador infiere que $Cheese \sqcup Vegetable \sqsubseteq Vegetarian$.

En este ejemplo hemos mostrado el caso ideal donde el planificador encuentra una solución al primer intento. Normalmente la búsqueda es mucho más compleja ya que las tareas se pueden resolver usando distintos operadores y métodos, y suelen existir distintas formas de satisfacer sus precondiciones. En el caso general el algoritmo de búsqueda necesitará hacer *backtracking* hasta encontrar una solución o explorar todo el espacio de posibilidades.

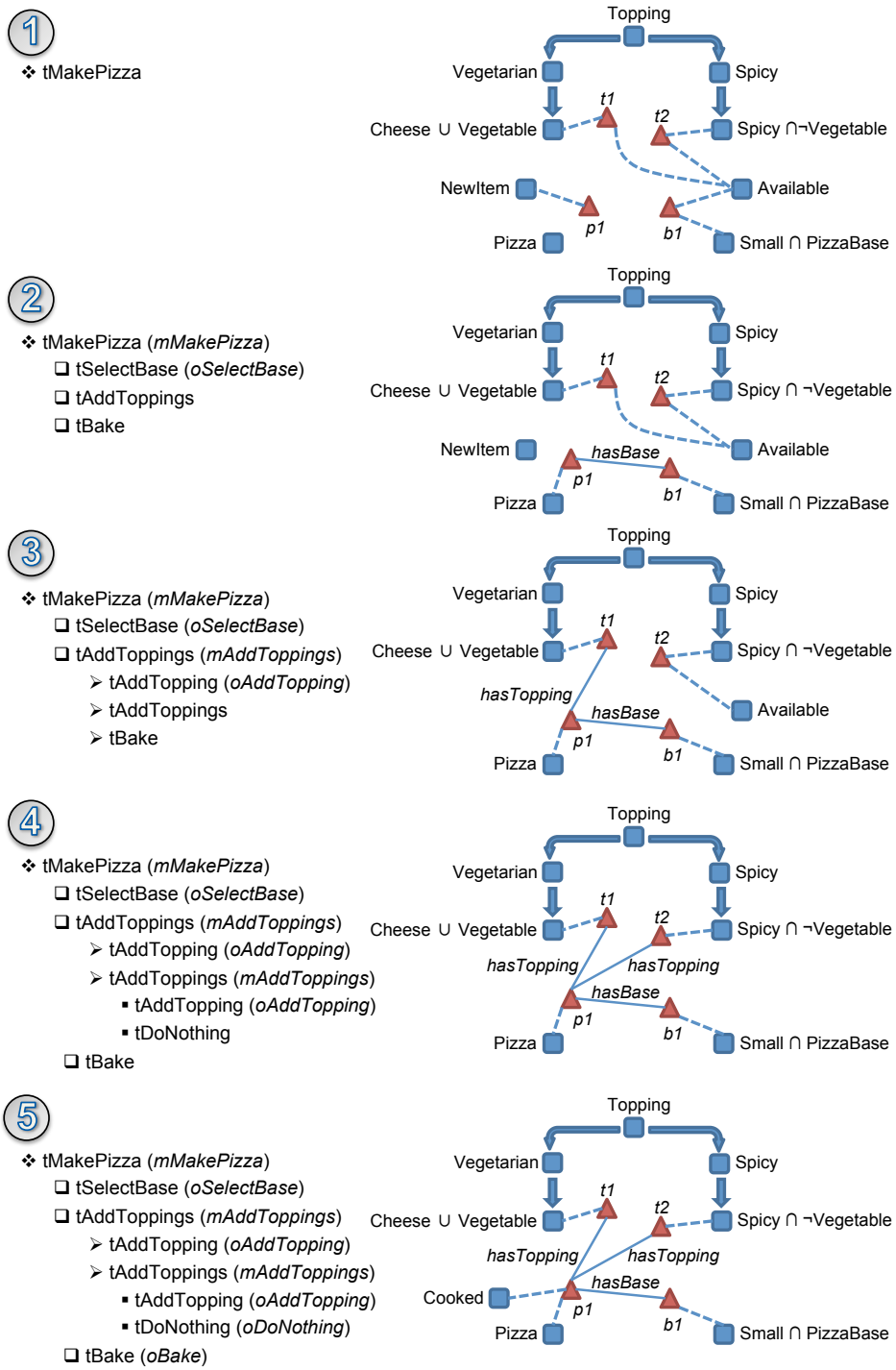


Figura 5.8: Búsqueda de la solución en un problema STN

5.6. Experimentos

Llegados a este punto parece obligado preguntarse cuál es el rendimiento de DLPlan si lo comparamos con otros planificadores. En los siguientes apartados vamos a describir algunos experimentos que hemos realizado, pero es importante que antes hagamos algunas reflexiones.

DLPlan es un planificador diseñado para trabajar en dominios ricos en conocimiento que se describen de manera natural usando ontologías. La gran ventaja de esta aproximación, precisamente, es que independizamos el conocimiento estático de un dominio del conocimiento específico de planificación. El conocimiento estático se representa de manera reutilizable usando expresivas ontologías que se pueden gestionar usando las potentes herramientas de la web semántica. Cualquier comparación con otro planificador implica tener que traducir el conocimiento almacenado en las ontologías a otro lenguaje, seguramente menos expresivo, con las dificultades que esto conlleva.

No estamos interesados tampoco en soluciones que, de manera manual, extraen el conocimiento de la ontología y lo reescriben de la forma más conveniente para el planificador. Aceptamos que dedicando el tiempo y los recursos necesarios cualquier base de conocimiento se puede adaptar a un planificador para que funcione de manera eficiente. Una de las ventajas de DLPlan es que podemos gestionar grandes ontologías de dominio sin necesidad de extraer previamente el conocimiento necesario para los problemas. El razonador DL, internamente, es capaz de determinar las partes de la base de conocimiento involucradas en los procesos de inferencia que le solicitamos e ignorar el resto. En un sistema de planificación ideal nos gustaría poder proporcionar toda la información sobre el dominio al sistema y dejar que él decida qué información es importante para los problemas que va a resolver.

El interfaz clásico (no jerárquico) de DLPlan recorre el espacio de estados de forma ciega, sin usar ninguna heurística. No tiene sentido, por tanto, realizar experimentos de rendimiento usando dominios no jerárquicos. Lo que vamos a hacer es comparar la versión STN con otros dos planificadores jerárquicos (JSHOP [102] y JSHOP2 [127]) resolviendo problemas muy sencillos en el dominio de las pizzas, que tiene bastante conocimiento. En concreto vamos a comparar cómo se comportan estos planificadores cuando aumenta el número de entidades en el estado inicial y el espacio de búsqueda que deben recorrer para alcanzar una solución.

5.6.1. Traducción de axiomas OWL-DL a SHOP

Para comparar DLPlan con los planificadores JSHOP necesitamos traducir el conocimiento del dominio al lenguaje que usan estos últimos. En SHOP se pueden definir un tipo de axiomas muy similares a los predicados derivados de PDDL. Un axioma SHOP es una expresión de la forma (:- a $L_1 \dots L_n$), donde a es un átomo lógico y cada L_i es una precondición lógica. El átomo a

se interpreta como cierto si alguno de los L_i se satisface en el estado actual.

El proceso de traducción que hemos utilizado es prácticamente igual al que se describe en la sección 3.2.2, y también sufre las mismas limitaciones que explicamos en su momento:

- Sólo podemos traducir la jerarquía conceptual básica. En general los axiomas OWL-DL $A \sqsubseteq C$ donde C es un concepto complejo no se pueden traducir.
- En los axiomas OWL-DL de equivalencia, que se usan para crear conceptos definidos, sólo se traduce una dirección de la doble implicación.
- Las restricciones sólo permiten detectar inconsistencias en el estado, no en el conocimiento del dominio.

A continuación mostramos 3 axiomas SHOP que surgen al traducir la ontología de la pizzería. El primero corresponde a traducir el axioma de inclusión $Meat \sqsubseteq PizzaTopping$, el segundo es la traducción del axioma de equivalencia $Spicy \equiv PizzaTopping \sqcap Hot$, y el último corresponde a traducir la restricción $Meat \sqsubseteq \neg Fish$ ($Meat$ y $Fish$ son disjuntos).

$$(:-(PizzaTopping\ ?x)\ ((Meat\ ?x))$$
$$(:-(Spicy\ ?x)\ ((PizzaTopping\ ?x)\ (Hot\ ?x\ ?z1)))$$
$$(:-(inc001)\ ((Meat\ ?x)\ (Fish\ ?x)))$$

Como en SHOP no existe un lenguaje de restricciones necesitamos simular este comportamiento. Para hacerlo añadiremos un nuevo axioma como el que mostramos a continuación, que se evalúa a cierto si hay alguna inconsistencia en el estado actual. Después añadimos la condición *(not (inconsistente))* a la precondición de todos los operadores y métodos. De este modo el planificador no va a alertarnos si llega a un estado inconsistente pero al menos no seguirá tratando de resolver el problema.

$$(:-\ (inconsistente)\ ((inc001))\ ((inc002))\ldots)$$

Al igual que pasaba con la traducción a PDDL, el dominio resultante es necesario revisarlo para comprobar que el conocimiento que se añade de forma implícita al *cerrar* la interpretación del mundo no afecta al tipo de problemas que vamos a proponer.

5.6.2. Problemas

Los problemas que hemos usado en los experimentos consisten en crear distintos tipos de pizzas a partir de unos ingredientes iniciales. Como los pla-

nificadores SHOP parten de estados iniciales concretos (no admiten incertidumbre), los problemas utilizan ingredientes concretos que se corresponden con los conceptos hoja de la ontología.

Hemos creado dos conjuntos de problemas para comprobar cómo se comportan los planificadores cuando: (1) aumentamos el número de objetos en el estado inicial, y (2) aumentamos el espacio de búsqueda que necesitan recorrer para hallar una solución. En el primer conjunto de problemas (*EG*), el objetivo consiste en construir siempre el mismo tipo de pizza pero en cada problema aumentamos el número de ingredientes y bases disponibles en el estado inicial (desde 10 a 200). En el segundo conjunto de problemas (*FIS*), utilizamos un número constante de ingredientes en el estado inicial e incrementamos de forma gradual el número de pizzas objetivo (desde 1 hasta 20).

Estamos interesados en comprobar cómo influye en el rendimiento el hecho de usar un razonador de DLs para gestionar el conocimiento estático. Para ello debemos tratar de eliminar otros factores que pueden influir en el experimento, como por ejemplo el orden en el que se recorre el espacio de búsqueda. En este caso todos los planificadores tratan de resolver las tareas usando los operadores / métodos en el mismo orden en el que aparecen en el fichero de descripción del dominio. El orden en el que se evalúan los operadores / métodos no es problemático porque definiremos los ficheros de dominio de manera equivalente. Donde sí podemos encontrar problemas es en la evaluación de las precondiciones. Si existen varias formas de satisfacer la precondición de un método / operador, dependiendo del orden en el que se prueben, es posible que haya que realizar *backtracking* más o menos veces hasta alcanzar una solución. En DLPlan no tenemos control sobre el orden en el que obtenemos las respuestas, ya que las precondiciones se evalúan como consultas DL sobre la base de conocimiento y Pellet no tiene un comportamiento determinista al evaluar consultas. Para evitar que el orden en el que recuperamos los ingredientes pueda afectar al espacio de búsqueda todos los problemas de hemos planteado se resuelven sin usar *backtracking*, es decir, cualquier sustitución que hace aplicable un operador / método lleva a una solución. De este modo sabemos que todos los planificadores van a recorrer el espacio de búsqueda de manera equivalente.

Además, todos los experimentos se ejecutan usando 3 versiones SHOP distintas del dominio de la pizzería. La versión *D1* es la más similar al dominio original y contiene axiomas SHOP para representar la jerarquía tipo-subtipo, para representar los conceptos definidos, y para comprobar las restricciones del dominio. En realidad, las restricciones del dominio son interesantes para comprobar la consistencia del estado inicial, pero no son necesarias durante el proceso de búsqueda si asumimos que el estado inicial es consistente y que los operadores nunca rompen la consistencia. La segunda versión del dominio, *D2*, contiene axiomas SHOP para representar la jerar-

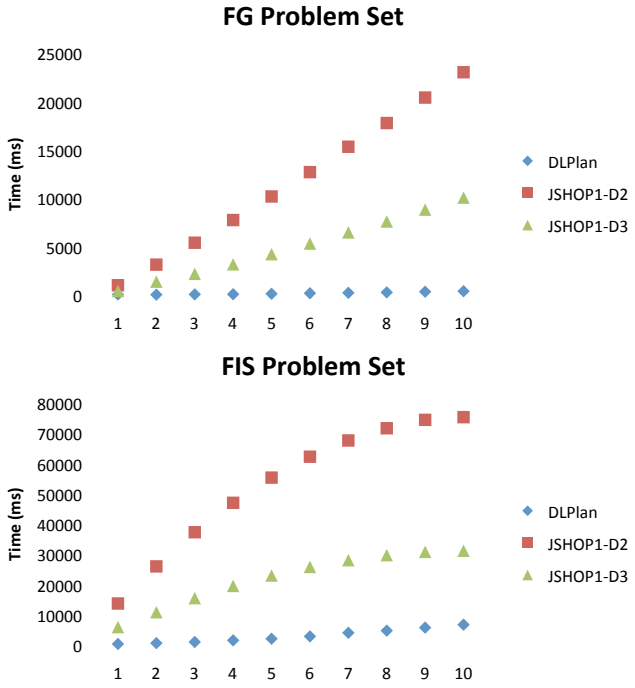


Figura 5.9: DLPlan y JSHOP

quía de tipos y los conceptos definidos pero no comprueba que los estados cumplan las restricciones del dominio. La última versión, *D3*, aún simplifica más la descripción del dominio eliminando de manera manual todos los axiomas que no son necesarios para resolver los problemas del experimento.

Por último, en los experimentos no hemos tenido en cuenta el tiempo necesario para clasificar la ontología del dominio en DLPlan, ni el tiempo necesario para compilar los dominios en JSHOP2, porque estas operaciones pueden realizarse *off-line* antes de empezar a resolver problemas. Cada problema se ha resuelto 4 veces y los valores que se muestran en las figuras 5.9 y 5.10 son la media de los tiempos obtenidos en cada problema.

5.6.3. JSHOP

La figura 5.9 muestra que DLPlan es mucho más rápido que JSHOP en todos los problemas. De hecho, los tiempos usando el dominio *D1* son tan altos que ni siquiera los mostramos en el gráfico. También es interesante comprobar los tiempos de JSHOP se ven muy afectados por el número de ingredientes que existen en el estado inicial. Hemos identificados dos razones que explican este mal comportamiento:

- Los átomos en la cabecera de los axiomas se vuelven a evaluar desde cero cada vez que se necesitan. De esta forma, al evaluar las precondiciones de los métodos y operadores se vueleven a evaluar los mismos axiomas muchas veces, aunque el estado no haya cambiado. DLPlan, por el contrario, usa un razonador que proporciona razonamiento incremental y sólo re-evalúa el conocimiento que se ve afectado por los cambios.
- Durante la búsqueda, todas las posibles descomposiciones de una tarea se calculan antes de aplicar ningún método. Esta aproximación es especialmente mala en problemas que se pueden resolver sin necesidad de *backtracking*. El algoritmo de búsqueda de DLPlan calcula de manera incremental nuevas maneras de resolver una tarea sólo cuando todas las anteriores han fallado.

La forma en la que el planificador gestiona el conocimiento del dominio y del estado actual resulta determinante para el rendimiento final. En este caso, DLPlan usa un razonador que gestiona este conocimiento de tipo ontológico de manera mucho más eficiente.

5.6.4. JSHOP2

JSHOP2 usa una novedosa técnica de compilación [105] que sintetiza planificadores específicos para un dominio a partir de la descripción de dicho dominio. Esto le permite realizar multitud de optimizaciones en el sistema final para mejorar el rendimiento. Sin embargo, esto también implica que el tiempo necesario para resolver un nuevo problema debe tener en cuenta el tiempo necesario para compilarlo. La figura 5.10 muestra el tiempo necesario para resolver los problemas sin tener en cuenta el tiempo de compilación de JSHOP2. Cuando usamos ficheros de problemas grandes (nuestros ficheros contenían unos 40 problemas) este tiempo es considerable, alrededor de 3 segundos por problema.

El gráfico superior muestra el comportamiento de ambos planificadores cuando se incrementa el número de ingredientes en el estado inicial. En este caso JSHOP2 puede manejar la versión completa del dominio *D1*. Podemos observar que el número de ingredientes en el estado inicial afecta más a DLPlan que a JSHOP2, pero si consideramos el tiempo de compilación que JSHOP2 requiere (los tiempos deberían incrementarse en 2 o 3 segundos) DLPlan todavía es una buena opción para problemas de tamaño medio.

El gráfico inferior muestra lo que ocurre cuando incrementamos el número de objetivos y, por tanto, el espacio de búsqueda. JSHOP2 presenta importantes problemas al utilizar la versión *D1* del dominio debido al número de axiomas que evalúa para comprobar la consistencia del estado. Esto nos lleva a pensar que la gestión de axiomas en JSHOP2 tampoco está muy optimizada. Sin embargo, usando la versión *D2* del dominio, que tiene muchos

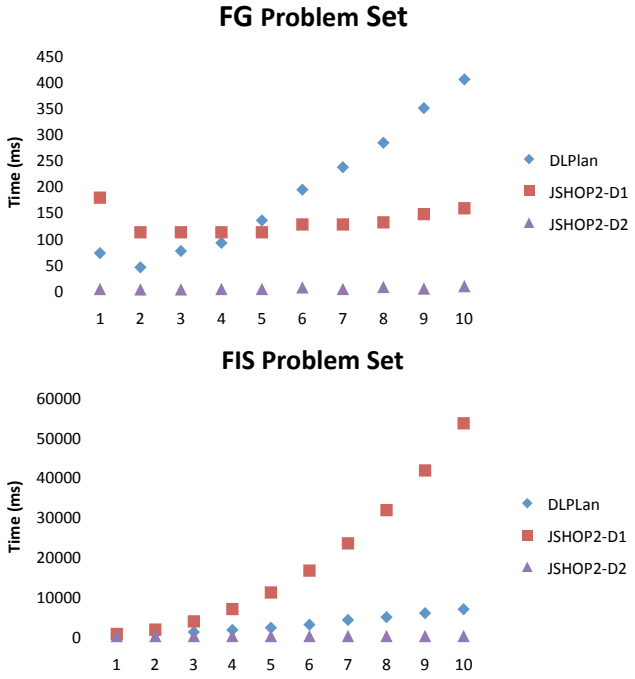


Figura 5.10: DLPlan y JSHOP2 (sin tiempos de compilación de problemas)

menos axiomas, resuelve todos los problemas muy rápido. JSHOP2 recorre el espacio de búsqueda mucho más rápido que DLPlan, lo que es natural porque asume mundo cerrado y el razonamiento es mucho más rápido. DLPlan parece ser una buena opción cuando el número de axiomas es muy alto o en problemas que no requieran una exploración de un espacio de búsqueda muy grande.

5.7. Disponibilidad del sistema

DLPlan es un proyecto de código libre desarrollado en Java que se distribuye usando la licencia AGPL. El sistema, junto la documentación y varios ejemplos de uso, se puede descargar desde `http://sourceforge.net/projects/dlplan/`.

5.8. Conclusiones

En este capítulo hemos descrito el sistema DLPlan, un planificador que utiliza ontologías OWL-DL para representar el conocimiento estático del dominio. Los ficheros que describen los dominios de planificación contienen

referencias a las ontologías que contienen la descripción del vocabulario y los axiomas. De este modo, tratamos de independizar, en la medida de lo posible, el conocimiento general del dominio del conocimiento específico de planificación (operadores, objetivos, ...). Además, esto nos permite seguir utilizando las herramientas de la web semántica para crear y gestionar ontologías, además de compartir las bases de conocimiento entre distintos sistemas inteligentes.

DLPlan utiliza un modelo de planificación muy simplificado en el que los operadores se interpretan de manera sintáctica. Esta aproximación aumenta la responsabilidad de los diseñadores que deben escribir los operadores de planificación, pero también permite actualizar el estado de manera eficiente. DLPlan delega en Pellet, un conocido razonador de DLs, la gestión del conocimiento estático y de ese modo es capaz de gestionar de manera eficiente dominios complejos con cientos o incluso miles de axiomas.

En este capítulo hemos descrito la arquitectura básica del sistema y hemos mostrado varios ejemplos de uso. Finalmente, hemos realizado algunos experimentos que parecen sugerir que DLPlan puede ser útil en dominios complejos donde la dificultad de los problemas reside en la gestión del conocimiento y no tanto en el recorrido inteligente del espacio de búsqueda.

Capítulo 6

Planificación basada en casos y ontologías

En el capítulo 4 describimos un modelo de planificación en el que usamos DLs para modelar el conocimiento. Esta aproximación tiene ciertas ventajas desde el punto de vista de la representación y gestión del conocimiento: (1) podemos usar modelos expresivos de tipo ontológico para representar dominios con una estructura y vocabulario complejos, (2) nos permite trabajar a partir de descripciones incompletas del estado y representar de forma compacta el espacio de búsqueda, (3) podemos detectar inconsistencias en la base de conocimiento de manera automática, etc. Pero toda esta flexibilidad tiene un precio: los procesos de razonamiento y búsqueda son muy costosos desde el punto de vista computacional [91].

Cuando construir planes desde cero resulta costoso, podemos tratar de aplicar aproximaciones basadas en casos en las que tratamos de aprovechar el esfuerzo invertido para resolver un problema cuando necesitamos resolver otro problema similar. De hecho, la capacidad de razonamiento con información incompleta de las DLs favorece la creación de planes válidos en un amplio abanico de situaciones. Recordemos, por ejemplo, el problema que planteábamos en la sección 4.1.3 en el que tratábamos de transformar una pizza vegetariana en una pizza con carne.

Comenzaremos el capítulo con unas breves reflexiones sobre el uso de DLs para representar, de manera estándar, conocimiento a distintos niveles de abstracción en sistemas CBR (del inglés *Case-Based Reasoning*) [2]. Después describiremos cómo generalizar problemas de planificación concretos (y sus soluciones) para construir casos abstractos que resulten útiles en otras situaciones distintas. También describiremos cómo almacenar los casos en una base de conocimiento DL y, de ese modo, aprovechar las capacidades de inferencia y el conocimiento del dominio a la hora de recuperarlos y adaptaros al contexto de los nuevos problemas [130].

Más tarde describiremos jCOLIBRI [111, 34], una plataforma diseñada

para facilitar la construcción de sistemas CBR. Esta plataforma ha sido desarrollada en nuestro grupo de investigación y goza de bastante popularidad en la comunidad CBR. Finalmente, describiremos la arquitectura de un planificador basado en casos (CBP, del inglés *Case-Based Planning*) como el que proponemos en este capítulo y su posible integración dentro de la plataforma jCOLIBRI.

6.1. Abstracción, DLs y casos

La abstracción ha jugado desde siempre un papel esencial en la representación del conocimiento [23]. La relación *is-a* aparece en el núcleo de los sistemas basados en marcos (*frames*) y redes semánticas (*semantic networks*), aunque no muy bien formalizada. Las lógicas descriptivas, como tecnología sucesora de estos sistemas, define una semántica formal bien fundada donde la abstracción en jerarquías conceptuales puede reducirse a problemas lógicos decidibles.

Las ontologías nos brindan la posibilidad de describir escenarios usando distinto nivel de detalle. Los conceptos de la parte superior representan tipos generales que se van especializando según descendemos por la jerarquía conceptual, hasta llegar al nivel de las instancias que representan entidades concretas de nuestro dominio. Esta riqueza de vocabulario nos permite describir tanto situaciones abstractas, en las que no conocemos o no nos interesan ciertos detalles, como situaciones concretas descritas con gran nivel de precisión.

Otra característica interesante de las DLs es que todas las inferencia realizadas a partir de una base de conocimiento seguirán siendo válidas si añadimos nueva información (consistente) a la base de conocimiento. Al añadir nuevo conocimiento estamos definiendo un escenario más específico, en el que, seguramente, será posible realizar nuevas inferencias, pero las relaciones entre entidades calculadas originalmente seguirán siendo válidas.

Por todo ello, concluimos que las DLs son una potente herramienta para representar conocimiento y razonar a distintos niveles de abstracción: permiten representar una terminología rica con la que describir situaciones con distinto nivel de detalle, y los razonamiento hechos a partir de una descripción abstracta de un escenario siguen siendo válidos en escenarios más concretos.

El concepto de abstracción también juega un papel importante en los sistemas CBR, aunque bajo distintos nombres como casos generalizados, prototipos o scripts. En [83] se propone una definición de caso abstracto ampliamente aceptada: un caso abstracto es un tipo particular de caso generalizado, en el sentido de que sustituye a un conjunto de casos concretos, que se representa usando algún formalismo simplificado (abstracto) en relación al que se usa para representar los casos concretos. Para trabajar con casos abs-

tractos es necesario disponer de conocimiento adicional sobre el dominio que permita construir casos abstractos a partir de casos concretos, recuperar los casos abstractos significativos para un nuevo problema, y adaptar los casos recuperados a las condiciones específicas del nuevo problema. Tradicionalmente, este conocimiento se ha representado usando taxonomías sencillas en las que un conjunto de valores (etiquetas) se abstraen usando su tipo (otra etiqueta). El conocimiento que se puede representar en estas taxonomías de etiquetas es muy limitado y, además, se representa usando mecanismos *ad-hoc* que dificultan su reutilización en otros sistemas y dominios.

En este capítulo proponemos el uso de ontologías DL como mecanismo expresivo y estándar para representar conocimiento a distintos niveles de abstracción. Usando esta representación declarativa para representar el conocimiento del dominio seremos capaces de: (1) generalizar el estado inicial en el que un plan es aplicable aumentando su uso potencial, (2) recuperar los planes relevantes para el nuevo problema, (3) adaptar los planes recuperados a la nueva situación inicial.

La idea básica tras nuestro planteamiento es que si un plan es capaz de alcanzar ciertos objetivos a partir de un estado inicial, también podemos utilizarlo para resolver problemas con un estado inicial más específico y objetivos más generales. Por ejemplo, si sabemos cómo ir en coche desde Madrid hasta Sevilla, también sabemos cómo salir de Madrid usando un Seat León. Debemos tener en cuenta que las soluciones que conseguimos al reutilizar planes, aunque válidas, pueden no ser muy buenas (seguramente la mejor manera de salir de Madrid en coche no sea conducir hasta Sevilla).

La limitación más importante de nuestra propuesta es que sólo somos capaces de reutilizar los planes si no es necesario modificar su estructura. Básicamente lo que hacemos es abstraer las propiedades de las entidades que intervienen en el plan usando la ontología del dominio, y buscar individuos en el nuevo estado inicial que cumplan dichas propiedades.

6.2. Planificación basada en casos usando DLs

Existen dos motivos básicos que pueden motivarnos a utilizar planificación basada en casos: mejorar la eficiencia y poder trabajar en dominios difíciles de modelar. La mejora de la eficiencia se basa en la idea intuitiva de que, a veces, es más fácil adaptar la solución que se utilizó para resolver un problema similar que tratar de crear una solución desde cero. Los casos también permiten trabajar en dominios que no somos capaces de modelar completamente porque pueden contienen conocimiento del dominio de manera implícita, conocimiento que no somos capaces de modelar de forma explícita. Pero eso también implica que debemos ser cuidadosos durante la fase de adaptación para no violar ninguna de la reglas del dominio implícitas en la base de casos.

La aproximación que vamos a explicar en este capítulo utiliza casos para tratar de mejorar la eficiencia. Es decir, supondremos que disponemos de un modelo del dominio suficientemente completo para resolver problemas usando un planificador generativo. Nuestro sistema CBP intenta resolver los problemas adaptando alguna de las soluciones de la base de casos, y si no lo consigue envía el problema al planificador generativo. Si el planificador generativo consigue resolverlo dispondremos de un nuevo caso que poder utilizar en el futuro.

En los siguientes apartados explicamos las distintas partes del sistema. Inicialmente la base de casos estará vacía, así que cuando llegue un nuevo problema usaremos el planificador generativo. Si el planificador generativo consigue resolverlo, dispondremos de un nuevo caso (par problema - solución) que trataremos de generalizar, usando la ontología del dominio, para conseguir un caso abstracto más versátil. Cada caso describe la situación inicial en la que es aplicable, el objetivo que alcanza y la secuencia de acciones que se debe aplicar para lograrlo.

A continuación debemos almacenar el caso en la base de casos. Como pretendemos recuperar los casos usando las capacidades de inferencia de un razonador, almacenaremos los casos en otra ontología indexados por objetivos. Cuando llegue un nuevo problema comprobaremos si alguno de los casos de la base de casos se puede aplicar: primero recuperaremos los casos con objetivos compatibles con el objetivo del problema actual, y después comprobaremos cuáles de ellos son aplicables en el nuevo estado inicial. Ambas operaciones se resuelven usando distintas consultas DL sobre las dos bases de conocimiento (la de los casos y la que representa el estado inicial) por lo que tanto la recuperación como la adaptación sacan partido de las capacidades de razonamiento inherente a estos lenguajes.

6.2.1. Ejemplo de problema - solución

Para describir el sistema CBP usaremos un ejemplo concreto basado en nuestro ya famoso dominio de la pizzería. Los operadores del dominio de planificación son los que mostramos a continuación. El operador *selectBase* selecciona una base y crea un nuevo objeto de tipo pizza, y los operadores *addToppingX* añaden un ingrediente a la pizza dependiendo del número de ingredientes original.

operator selectBase (?p, ?b)
pre: NewItem(?p), Base(?b), Available(?b)
post: not Available(?b), not NewItem(?p),
hasBase(?p, ?b), PizzaWith0Toppings(?p)

operator addTopping0 (?p, ?t)
pre: Topping(?t), Available(?t), PizzaWith0Toppings(?p)
post: not Available(?t), not PizzaWith0Toppings(?p),

hasTopping(?p,?t), PizzaWith1Toppings(?p)

operator addTopping1 (?p, ?t)
pre: Topping(?t), Available(?t), PizzaWith1Toppings(?p)
post: not Available(?t), not PizzaWith1Toppings(?p),
hasTopping(?p,?t), PizzaWith2Toppings(?p)

El objetivo del problema consiste en crear una pizza margarita con masa fina. Recordemos que, según la ontología, una pizza margarita es un tipo de pizza que contiene dos ingredientes: tomate y queso. El objetivo de nuestro problema es el siguiente:

$$Margherita(?x), hasBase(?x, ?y), ThinBase(?y)$$

A continuación mostramos el estado inicial del problema, en el que disponemos de los siguientes ingredientes:

$$NewItem(p1), Available(b1), ThinBase(b1), Available(t1),$$
$$SundriedTomato(t1), Available(t2), Mozzarella(t2), Available(t3),$$
$$VegetarianTopping(t3), SpicyTopping(t3), Mushrooms(t4)$$

Los ingredientes que nos interesan para resolver este problema son *b1*, por se la única base fina disponible, y *t1* y *t2* que son tipos particulares de tomate y queso. Usando toda esta información el planificador generativo puede construir el siguiente plan que resuelve el problema: seleccionar la base *b1*, añadir el ingrediente *t1* y añadir el ingrediente *t2*.

$$selectBase(p1, b1),\ addTopping0(p1, t1),\ addTopping1(p1, t2)$$

En el siguiente apartado veremos cómo podemos transformar este par problema - solución en un caso. Además, explicaremos cómo generalizar el estado inicial y, de ese modo, ampliar los escenarios en los que el caso puede resultar útil.

6.2.2. Generación de casos

Los casos representan episodios de resolución de un problema. En nuestra aproximación los casos tienen una estructura muy sencilla: estado inicial, objetivos y plan. Lo que queremos representar en cada caso es que aplicando el plan desde ese estado inicial alcanzamos otro estado en el que se cumplen los objetivos. Desde este punto de vista cada par problema - solución puede verse como un caso concreto.

Definición 23 (Caso concreto) *Dado un problema de planificación* $\mathcal{P} = (\mathcal{T}, \mathcal{A}, \mathcal{O}, G)$ *y un plan* π *que lo resuelve, el vector* $C = (\mathcal{A}, G, \pi)$ *es un caso concreto que representa al problema* $\mathcal{P}$*.*

Sin embargo, este tipo de casos concretos no tiene mucha utilidad, ya que usa individuos específicos en el estado inicial y en el plan solución. Para aumentar la usabilidad del caso necesitamos sustituir los individuos con variables nuevas.

Definición 24 (Caso) *Un caso es una generalización de un caso concreto que resulta de sustituir los individuos del caso concreto con variables nuevas y se representa mediante un vector* $(pre, post, plan)$ *donde* pre *es una consulta conjuntiva DL que representa la precondición del caso,* $post$ *es otra consulta conjuntiva que representa la postcondición del caso, y* $plan$ *es una generalización de un plan que puede usar variables.*

La precondición del caso indica las condiciones que se deben dar en el estado inicial para que el caso sea aplicable. Al utilizar la precondición como consulta sobre la KB que representa el estado inicial de un nuevo problema conseguiremos sustituciones que asociarán las variables del plan generalizado con individuos concretos, produciendo un plan ejecutable. La postcondición del caso, a su vez, representa los objetivos que nos garantiza la ejecución del plan a partir de un estado inicial que cumpla la precondición.

Usando todas estas ideas ya podríamos generar un caso a partir del problema y la solución que describimos en el apartado anterior. Sin embargo, si lo hiciéramos directamente conseguiríamos una caso con una precondición demasiado restrictiva. Por ejemplo, en el estado inicial del problema existía un ingrediente *t3* vegetariano y picante, que trasladaría a la precondición del caso la exigencia de que exista uno de estos ingredientes, aunque ese ingrediente en particular no se usa para nada en el plan solución.

Por tanto, antes de generar el caso trataremos de generalizar el estado inicial eliminando toda la información que no sea necesaria para resolver el problema. El algoritmo que mostramos a continuación no se limita a eliminar los asertos sobre individuos que no aparecen en la solución, también utiliza la ontología del dominio para generalizar, cuando sea posible, los tipos de los individuos que sí intervienen en el plan.

Dados un problema de planificación $\mathcal{P} = (\mathcal{T}, \mathcal{A}, \mathcal{O}, G)$ y su plan solución π podemos generalizar el estado inicial usando los siguientes pasos:

1. añadir explícitamente al estado inicial todos los asertos α de la ABox tales que $(\mathcal{T}, \mathcal{A}) \models \alpha$.

2. eliminar uno a uno los asertos ABox del estado inicial que no son necesarios para resolver el problema usando π.

El primer paso añade de manera explícita al estado inicial todos los asertos ABox que son consecuencia lógica de la base de conocimiento. Por tanto, después de este paso obtenemos una nueva base de conocimiento semánticamente equivalente a la anterior. El segundo paso consiste en eliminar asertos

que no son necesarios para resolver el problema (eliminación sintáctica), pero ahora, como resultado del paso anterior, la eliminación de un aserto no implica perder todo el conocimiento que se infería a partir de él.

Para realizar esta generalización necesitamos comprobar si el plan sigue siendo válido cada vez que eliminamos información del estado inicial. La forma más sencilla, aunque costosa, es ejecutar el plan tras cada eliminación. Analizando el plan, los objetivos y el estado inicial seguramente podríamos realizar esta generalización de manera más inteligente pero, para simplificar, asumiremos que la construcción de casos abstractos se realiza *off-line* y el rendimiento no es un problema.

Otro detalle interesante es que, dependiendo del orden en el que eliminemos los asertos, podemos llegar a distintos estados iniciales mínimos a partir de los cuales el plan sigue siendo válido. Nosotros asumiremos que la eliminación de asertos se realiza de manera no determinista, pero en un sistema real habría que tener en cuenta que la estrategia de eliminación utilizada puede afectar al rendimiento final del sistema.

A continuación vamos a ver cómo aplica todo esto a la generación de un caso a partir del ejemplo del apartado anterior. En primer lugar debemos añadir al estado inicial todos los asertos ABox que se pueden inferir a partir de la base de conocimiento. En realidad lo que conseguimos de este modo es una ABox equivalente a la inicial en la que aparecen de manera explícita todos los tipos y relaciones entre los individuos.

$$
\begin{aligned}
&NewItem(p1), Available(b1), ThinBase(b1), PizzaBase(b1),\\
&AvailableBase(b1), Available(t1), SundriedTomato(t1), Tomato(t1),\\
&Vegetable(t1), VegetarianTopping(t1), Topping(t1), AvailableTopping(t1),\\
&Available(t2), Mozarella(t2), Cheese(t2), VegetarianTopping(t2),\\
&Topping(t2), AvailableTopping(t2), Available(t3), VegetarianTopping(t3),\\
&SpicyTopping(t3), Topping(t3), AvailableTopping(t3), Mushrooms(t4),\\
&Vegetable(t4), VegetarianTopping(t4), Topping(t4)
\end{aligned}
$$

Después vamos eliminando, uno por uno, todos aquellos asertos que no son necesarios para ejecutar el plan y garantizar que llegamos a un estado en el que se cumplen los objetivos del problema. El estado inicial se queda con los siguiente asertos:

$$
\begin{aligned}
&NewItem(p1), Available(b1), ThinBase(b1),\\
&Available(t1), Tomato(t1), Available(t2), Cheese(t2)
\end{aligned}
$$

Finalmente usamos este nuevo estado inicial para construir un caso abstracto. Intuitivamente, el conocimiento representado en este caso es el siguiente: para construir una pizza margarita con base fina lo único que nece-

sita es una base fina y dos ingredientes de tipo tomate y queso; coge la base, añade el tomate y finalmente añade el queso.

pre: NewItem(?x1), Available(?x2), ThinBase(?x2),
Available(?x3), Tomato(?x3), Available(?x4), Cheese(?x4)
post: Margherita(?x), hasBase(?x,?y), ThinBase(?y)
plan: selectBase(?x1, ?x2), addTopping0(?x1, ?x3),
addTopping1(?x1, ?x4)

6.2.3. Base de casos

Necesitamos estructurar la base de casos para poder recuperar los casos *relevantes* de manera eficiente. Consideramos que un caso es relevante para resolver un problema si su precondición se satisface en el estado inicial del problema y su postcondición es más específica que el objetivo del problema. Al exigir que la precondición se cumpla en el estado inicial del problema nos aseguramos de que el plan almacenado en el caso es ejecutable en el nuevo contexto. Si la postcondición del caso es más específica que el objetivo del problema significa que al ejecutar el plan alcanzamos un estado en el que se cumple una condición más restrictiva que la impuesta por el problema y, por tanto, los objetivos del problema también se satisfacen.

Almacenaremos los casos en una nueva base de conocimiento para poder aprovechar las capacidades de inferencia de un razonador durante el proceso de recuperación. La TBox de esta base de conocimiento contendrá todo el vocabulario del dominio (axiomas de la TBox de la ontología del dominio), además de un nuevo concepto *Case* para representar los casos y un nuevo rol *hasSK* para relacionar cada caso con su información. En particular, almacenaremos los casos indexados por sus objetivos, sustituyendo las variables que aparecen en el objetivo con nuevos individuos *skolem*, y usando el rol *hasSK* para relacionar cada caso con sus individuos.

Tendremos, por tanto, dos bases de conocimiento en el sistema: una para representar el estado y otra para almacenar los casos. Ambas KB comparten la misma terminología (a excepción del concepto *Case* y el rol *hasSK*), pero la ABox de la primera se usa para representar el estado actual del planificador y la ABox de la segunda para representar los casos.

Veamos un ejemplo concreto. El caso que construimos en el apartado anterior podemos almacenarlo en la base de casos usando los siguientes axiomas ABox. La primera línea es el resultado de reemplazar las variables en la postcondición con nuevos individuos skolem *sk1* y *sk2*, y la segunda línea indica que esta información pertenece al caso *c1*.

$$Margherita(sk1), hasBase(sk1, sk2), ThinBase(sk2)$$
$$Case(c1), hasSK(c1, sk1), hasSK(c1, sk2)$$

Supongamos que ahora tenemos que resolver un nuevo problema que consiste en crear una pizza vegetariana con masa fina. Podemos recuperar los casos cuyo objetivo es más específico usando la consulta siguiente. El individuo *c1* se asociará a la variable *?z* porque el razonador infiere que una pizza margarita es un tipo de pizza vegetal ya que todos sus ingredientes son de tipo vegetariano.

$$VegetarianPizza(?x), hasBase(?x, ?y), ThinBase(?y)$$
$$Case(?z), hasSK(?z, ?x), hasSK(?z, ?y)$$

Una vez hemos recuperado todos los casos con una postcondición compatible con el objetivo del nuevo problema, debemos determinar cuáles de ellos son aplicables desde el nuevo estado inicial. Para hacerlo utilizaremos la precondición del caso como consulta sobre el estado inicial del problema. Si la consulta tiene alguna respuesta (cada respuesta es una sustitución) significa que el plan es ejecutable usando dicha sustitución.

6.2.4. Recuperación y adaptación

En realidad ya hemos explicado el proceso de recuperación en el apartado anterior. Recordemos que un caso $C = (pre, post, plan)$ representa la siguiente información: si ejecutamos el plan a partir de un estado que cumple la precondición, alcanzaremos otro estado en el que se cumple la postcondición. Por tanto, cuando tenemos que resolver un nuevo problema $\mathcal{P} = (\mathcal{T}, \mathcal{A}, \mathcal{O}, G)$ en el que el estado inicial es $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ necesitamos recuperar los casos en los que la consulta precondición tiene alguna respuesta en el nuevo estado inicial ($pre(\mathcal{K}) \neq \emptyset$) y cuya postcondición es más restrictiva que el objetivo del problema ($post \sqsubseteq_{\mathcal{T}} G$).

Supongamos, por ejemplo, que tenemos que resolver un nuevo problema que tiene como objetivo construir una pizza vegetariana a partir de los siguiente ingredientes:

$$NewItem(np1), ThinBase(nb1), Available(nb1), SlicedTomato(nt1),$$
$$Available(nt1), Parmesan(nt2), Available(nt2), Beef(nt3), Available(nt3)$$

Antes de intentar resolver este problema usando el planificador generativo vamos a comprobar si tenemos algún caso en la base de casos cuya solución podamos reutilizar. Para ello, comenzamos recuperando los casos cuya postcondición es más restrictiva que el objetivo de nuestro problema, es decir, casos que permitan construir una pizza vegetariana. Podemos recuperar todos estos casos, usando las capacidades de inferencia del razonador, a través de la siguiente consulta:

$$Caso(?x), hasSK(?x, ?y), VegetarianPizza(?y)$$

Esta consulta recuperará el caso *c1* que almacenamos en el apartado anterior, porque una pizza margarita de base fina es un tipo de pizza vegetariana.

Ahora debemos comprobar si la precondición del caso *c1* se satisface en el nuevo estado inicial. Recordemos que la precondición de este caso es la siguiente consulta:

$$NewItem(?x1), Available(?x2), ThinBase(?x2),$$
$$Available(?x3), Tomato(?x3), Available(?x4), Cheese(?x4)$$

Que se satisface en el nuevo estado inicial usando la siguiente sustitución: $\{?x1 = np1, ?x2 = nb1, ?x3 = nt1, ?x4 = nt2\}$. Usando esa sustitución sobre el plan almacenado en el caso obtenemos un nuevo plan solución para nuestro problema:

$$selectBase(np1, nb1),\ addTopping0(np1, nt1),\ addTopping1(np1, nt2)$$

Es importante resaltar que, al utilizar consultas DL para recuperar los casos relevantes para el nuevo problema, estamos adaptando los casos de manera implícita. Y todas estas adaptaciones se basan en inferencias realizadas a partir del conocimiento almacenado en la ontología del dominio. Por ejemplo, si comparamos el problema original a partir del cual obtuvimos el caso *c1* con el nuevo problema que acabamos de resolver, nos daremos cuenta de que: (1) hemos reutilizado un plan creado para construir un tipo de pizza en un problema en el que necesitábamos construir otro tipo de pizza más abstracto, y (2) durante la generalización y especialización del caso hemos sustituido implícitamente un ingrediente de tipo *SundriedTomato* con otro de tipo *SlicedTomaro*, y un ingrediente de tipo *Mozzarella* con otro de tipo *Parmesan*.

6.2.5. Arquitectura del sistema

La figura 6.1 muestra la arquitectura básica que tendría este planificador. Cuando llega un nuevo problema, el subsistema de planificación basada en casos intenta recuperar algún caso relevante y adaptar su solución. El proceso de recuperación se hace en dos fases: primero se recuperan los casos con objetivos compatibles, y después se comprueba cuáles de ellos son ejecutables desde el nuevo estado inicial. La primera fase se resuelve usando una consulta sobre la KB que almacena los casos, y la segunda ejecutando tantas consultas como casos hayamos recuperado en la fase anterior sobre la KB que representa el estado inicial. Si todo va bien habremos conseguido adaptar un caso y tendremos una solución válida al problema.

Si alguna de las dos fases de la recuperación falla debemos tratar de resolver el problema usando el subsistema de planificación generativa. Para

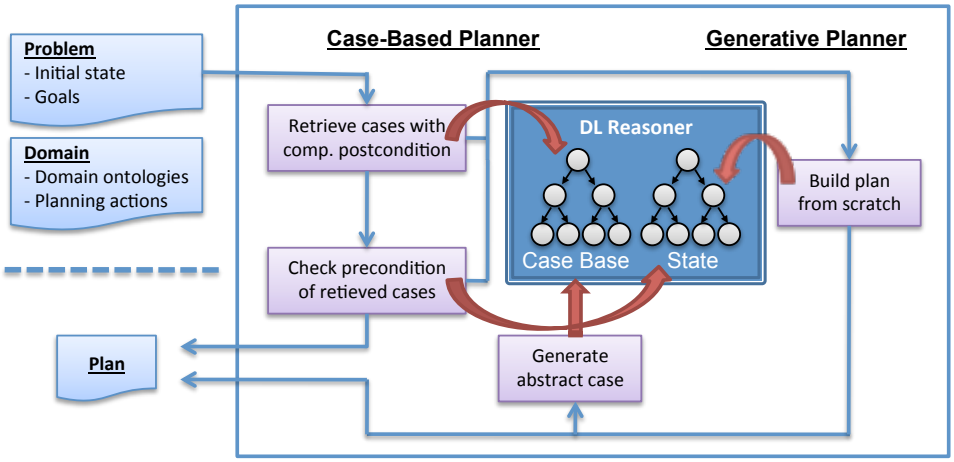


Figura 6.1: Arquitectura del planificador mixto

resolver el problema, este sistema necesitará acceder a la KB que representa el espacio de búsqueda. Si conseguimos un plan solución podemos almacenarlo para posteriormente generalizarlo y añadir un nuevo caso abstracto a la base de casos.

Ambos subsistemas de planificación necesitan acceder al conocimiento del dominio. En el diagrama hemos dibujado un único razonador que gestiona las dos KBs y es accesible desde ambos planificadores. Debemos tener en cuenta que las dos KBs prácticamente comparten la misma TBox así que usando un único razonador seguramente podamos realizar ciertas optimizaciones.

6.3. jCOLIBRI

jCOLIBRI [111, 34] es una versátil plataforma para el desarrollo de aplicaciones CBR desarrollada en nuestro grupo de investigación a lo largo de los últimos años [116, 117]. La plataforma consta de un arquitectura modular y extensible, y de distintos componentes y métodos ya implementados que facilitan el desarrollo de nuevas aplicaciones.

Comenzaremos describiendo las principales características de esta plataforma y la funcionalidad básica que nos ofrece. Más tarde, describiremos cómo podríamos implementar un planificador basado en casos como el que hemos descrito en este capítulo extendiendo las clases que proporciona esta plataforma.

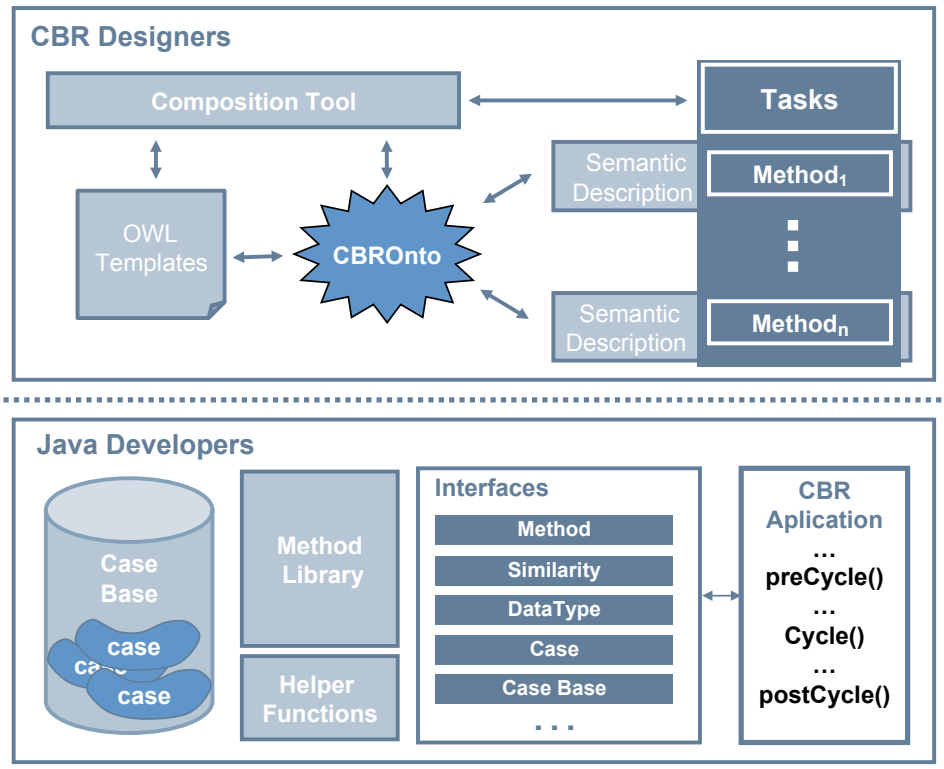


Figura 6.2: Arquitectura multicapa de jCOLIBRI

6.3.1. Arquitectura

jCOLIBRI está orientado tanto a desarrolladores de sistemas como a diseñadores sin conocimiento de programación. Para satisfacer las necesidades de ambos tipos de usuarios, la plataforma utiliza una arquitectura con dos capas bien diferenciadas (figura 6.2), cada una de ellas orientadas a un perfil de usuario distinto.

La capa inferior está orientada a los desarrolladores y es un armazón de caja blanca que contiene los componentes básicos para implementar distintos tipos de sistemas CBR: gestión de la persistencia de los casos, representación y organización en memoria de los casos, etc. Esta capa ofrece, además, un marco integrador que facilita la incorporación de nuevos métodos y técnicas CBR, promoviendo que los desarrolladores compartan los algoritmos que implementan. El armazón incorpora actualmente dos extensiones que proporcionan métodos para el desarrollo de aplicaciones de CBR textual y aplicaciones CBR con conocimiento intensivo. Finalmente, el armazón también ofrece la funcionalidad necesaria para evaluar, mantener e incluso visualizar los sistemas desarrollados.

En las aplicaciones de CBR textual los casos contienen textos por lo que

es necesario recurrir a técnicas especiales para el cálculo de similitud y el proceso de adaptación. jCOLIBRI ofrece dos tipos de métodos: los semánticos [115], que intentan capturar el significado del texto y representarlo de manera estructural para poder aplicar técnicas genéricas de CBR, y los estadísticos [112], que aplican conocidos algoritmos estadísticos del área de la recuperación de información.

En las aplicaciones CBR con conocimiento intensivo, el conocimiento almacenado en los casos se complementa con conocimiento general del dominio en forma de ontologías. El uso de ontologías es un mecanismo estándar e intuitivo para representar conocimiento que fomenta su reutilización en distintas aplicaciones. El uso de ontologías en jCOLIBRI [114, 35] permite almacenar los casos utilizando estructuras semánticas, utilizar similitudes entre casos que explotan la estructura del dominio, aumentar la expresividad de las consultas del usuario, usar conocimiento extra en la fase de adaptación, etc.

La capa superior de la arquitectura está orientada a usuarios diseñadores e incluye un conjunto de herramientas que permite crear sistemas CBR a través de interfaces gráficos sin necesidad de conocer los detalles del código. El proceso de autoría se realiza mediante plantillas que representan distintos tipos de sistemas CBR y que se pueden instanciar utilizando distintos componentes de la capa de programación. Además, tanto las plantillas como los componentes software se describen utilizando etiquetas semánticas de manera que el sistema puede razonar y ayudar al usuario durante el proceso de instanciación de las plantillas en sistemas CBR concretos [113, 136].

Nosotros nos vamos a centrar en la capa orientada a desarrolladores. Concretamente en las aplicaciones CBR de conocimiento intensivo que utilizan ontologías para representar conocimiento general del dominio con el que completar el conocimiento almacenado en los casos [114]. jCOLIBRI contiene un módulo llamado *OntoBridge* que facilita la comunicación con un razonador OWL-DL ofreciendo un interfaz sencillo, aunque limitado, para realizar operaciones comunes con ontologías: clasificación, recorrido de la jerarquía conceptual, etc.

6.3.2. Funcionalidad

La figura 6.3 muestra la funcionalidad básica que ofrece la plataforma organizada en distintas categorías.

Persistencia

La persistencia de los casos en JCOLIBRI se lleva a cabo mediante conectores, componentes software que tratan de independizar la aplicación del medio de persistencia utilizado. Actualmente existen conectores que permiten almacenar los casos en ficheros de texto, bases de datos relacionales y

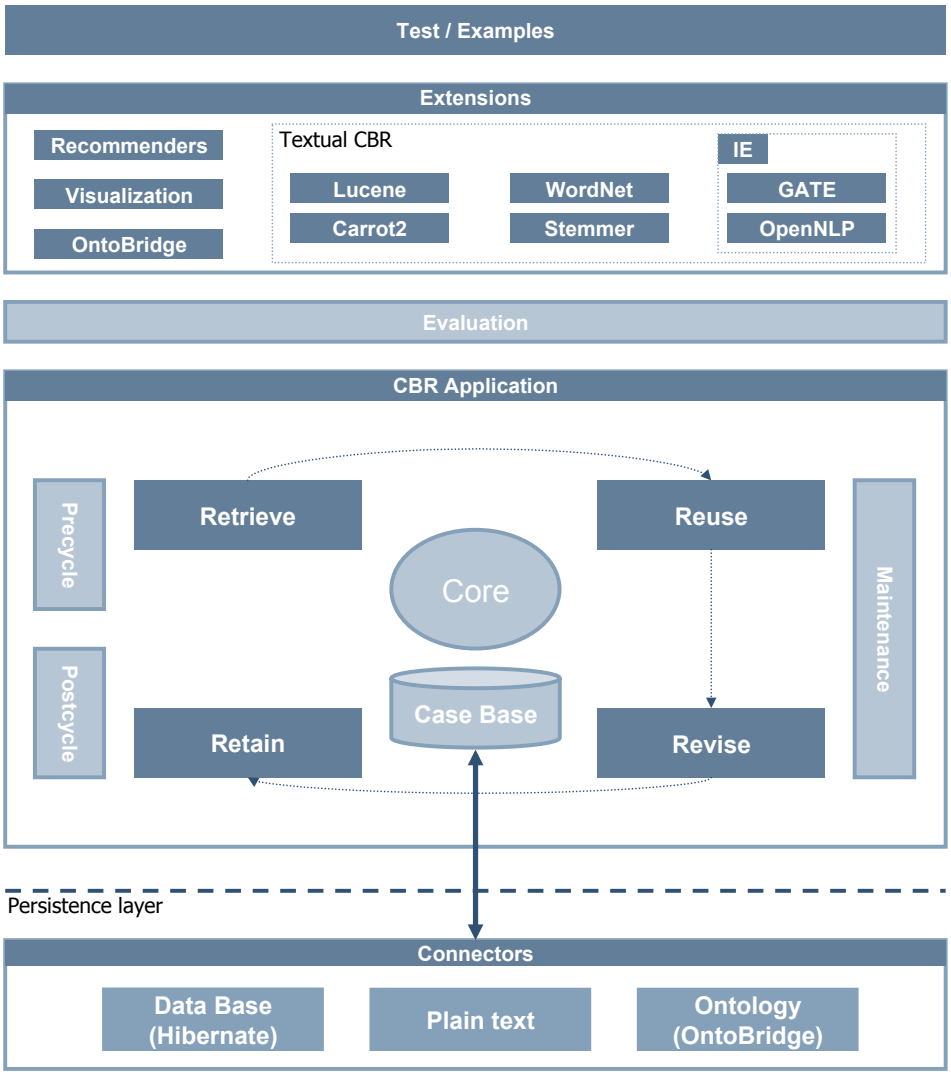


Figura 6.3: Organización de la funcionalidad de jCOLIBRI

ontologías.

El conector de bases de datos está implementado usando Hibernate [104], un potente armazón que se utiliza en multitud de aplicaciones J2EE para gestionar la persistencia. Hibernate no sólo independiza del sistema gestor de bases de datos que se esté utilizando, también facilita las conversiones entre el modelo orientado a objetos de las aplicaciones Java y el modelo relacional de las bases de datos. Para utilizar el conector de bases de datos en jCOLIBRI se utiliza un fichero XML que asocia los atributos de los casos con distintas columnas de las tablas de la base de datos.

El conector de ontologías permite aprovechar la estructura semántica de la ontología para organizar los casos, y las capacidades de inferencia de los razonadores de lógicas descriptivas para recuperar casos usando consultas semánticas. La idea básica consiste en almacenar los casos en la ontología usando mapas de individuos y relaciones, y dejar que el razonador gestione la información y resuelva las consultas. Cuando los casos son muy grandes es posible utilizar una aproximación mixta donde parte del caso se representa en la ontología, para poder hacer recuperación semántica, pero el resto del caso se almacena en una base de datos relacional.

Núcleo

El núcleo contiene las clases más básicas del armazón y los interfaces que definen los distintos elementos que encontramos en los sistemas CBR: *Connector*, *CBRCaseBase*, *CBRQuery*, *CBRCase*, *CaseComponent*, *Attribute*, etc. Este paquete no se debe modificar ya que cualquier cambio afectaría al resto del sistema. Los desarrolladores que quieran compartir sus métodos con el resto de la comunidad deben utilizar el sistema de extensiones que explicamos más adelante.

Base de casos

Los sistemas CBR pueden utilizar distintas estrategias a la hora de gestionar el acceso a la base de casos. La base de casos puede estar cargada total o parcialmente en memoria, por lo que puede resultar necesario utilizar sistemas de *cache*. A su vez, los casos cargados en memoria se pueden organizar utilizando distintas estructuras para resolver la fase de recuperación de manera eficiente. Actualmente, jCOLIBRI sólo proporciona dos estructuras básicas para recuperar casos: acceso secuencial o asociativo por un campo identificador.

Métodos de recuperación

Uno de los métodos más utilizados en los sistemas CBR es la recuperación k-NN (*Nearest Neighbors*) que recupera los k casos más próximos a la

consulta. Para definir el nivel de proximidad, la consulta se compara con los casos de la base de casos usando alguna medida de similitud. En jCOLIBRI existen muchas medidas de similitud, tanto globales como locales, para comparar atributos de distintos tipos: enumerados, cadenas de texto, números, estructuras, etc. También se proporcionan medidas de similitud especializadas para sistemas de CBR textual, sistemas recomendadores, y sistemas que usan ontologías para representar el conocimiento del dominio.

Además de la recuperación k-NN, también se proporcionan otros métodos de recuperación que pueden resultar interesantes en ciertos tipos de aplicaciones: recuperación por diversidad basada en la media, recuperación por medio de filtros, técnicas estadísticas para determinar la relevancia de los textos, etc.

Métodos de adaptación y revisión

Estas dos etapas son muy dependientes de la aplicación, por lo que jCOLIBRI sólo incorpora métodos básicos para copiar la solución de un caso a la consulta, copiar valores de los atributos desde la descripción a la solución, o calcular proporciones directas entre los valores de los atributos.

Mantenimiento

Estos métodos permiten reducir el tamaño de la base de casos eliminando aquellos casos redundantes o poco significativos. Algunos de los algoritmos implementados son: BBNR (*Blame-based Noise Reduction*), CRR (*Conservative Redundancy Removal*), RENN (*Repeated Edited Nearest Neighbor*), RC (*Relative Cover*) o ICF (*Iterative Case Filtering*).

Evaluación

Actualmente existen los siguientes métodos de evaluación implementados:

- *Leave One Out.* Consiste en extraer un caso de la base de casos y utilizarlo como consulta.
- *Hold Out.* Divide la base de casos en dos conjuntos: los casos que se van a utilizar como consultas y los que van a formar la base de casos en la evaluación.
- *N-Fold.* Divide la base de casos en varios conjuntos. En las distintas iteraciones de la evaluación un conjunto de casos se utiliza como consultas y los restantes como base de casos.

Además, jCOLIBRI proporciona herramientas visuales para configurar la evaluación y ver los resultados de la misma de manera visual usando distintas gráficas.

Sistema de extensiones

Las extensiones empaquetan componentes, métodos y medidas de similitud útiles para desarrollar sistemas CBR específicos. jCOLIBRI incluye extensiones para desarrollar sistemas de CBR textual, aplicaciones CBR con conocimiento intensivo y sistemas de recomendación. También se pueden desarrollar extensiones que aporten funcionalidades nuevas a la plataforma aplicables a distintos tipos de sistemas CBR.

6.3.3. Planificación basada en casos en jCOLIBRI

jCOLIBRI pretende ser una plataforma neutra para construir distintos tipos de sistemas CBR. La funcionalidad que ofrece la plataforma gira en torno al conjunto de métodos y estructuras de datos implementados en el armazón, que va aumentando a medida que los desarrolladores implementan nuevos sistemas con nuevos requisitos. Actualmente, la mayor parte los métodos disponibles están relacionados con el desarrollo de sistemas CBR de recomendación y sistemas CBR textuales, por lo que no encontraremos demasiada ayuda para implementar un planificador basado en casos.

De todas formas, a continuación describimos de manera concisa las clases que tendríamos que extender para desarrollar un sistema CBP como el que hemos descrito en este capítulo. En los siguientes apartados haremos referencia a las distintas clases que aparecen en el diagrama de la figura 6.4.

Representación de los casos

Los casos en jCOLIBRI [111, 135] contienen 4 campos: la descripción del problema, su solución, una posible justificación de la solución propuesta, y el resultado que se obtiene al aplicar dicha solución (podría fallar). Cada uno de estos 4 atributos se debe implementar como una clase *JavaBean* que extiende el interfaz *CaseComponent* y que proporciona métodos *get* y *set* para acceder a sus atributos. Usando esta infraestructura jCOLIBRI es capaz de gestionar la persistencia de los casos usando conectores.

Los casos que necesitamos en el planificador sólo van a utilizar los campos descripción y solución. La descripción del caso (clase *CaseDescription*) contiene dos consultas conjuntivas DL que representan su precondición y postcondición. La solución del caso (clase *CaseSolution*) contiene el plan que resuelve el problema. Además, como todos estos atributos se van a leer y escribir usando un conector, deben implementar el interfaz *TypeAdaptor* que establece un interfaz común para la persistencia.

Base de casos

En nuestro sistema la base de casos es una base de conocimiento DL gestionada por un razonador. Los casos podemos tenerlos inicialmente en

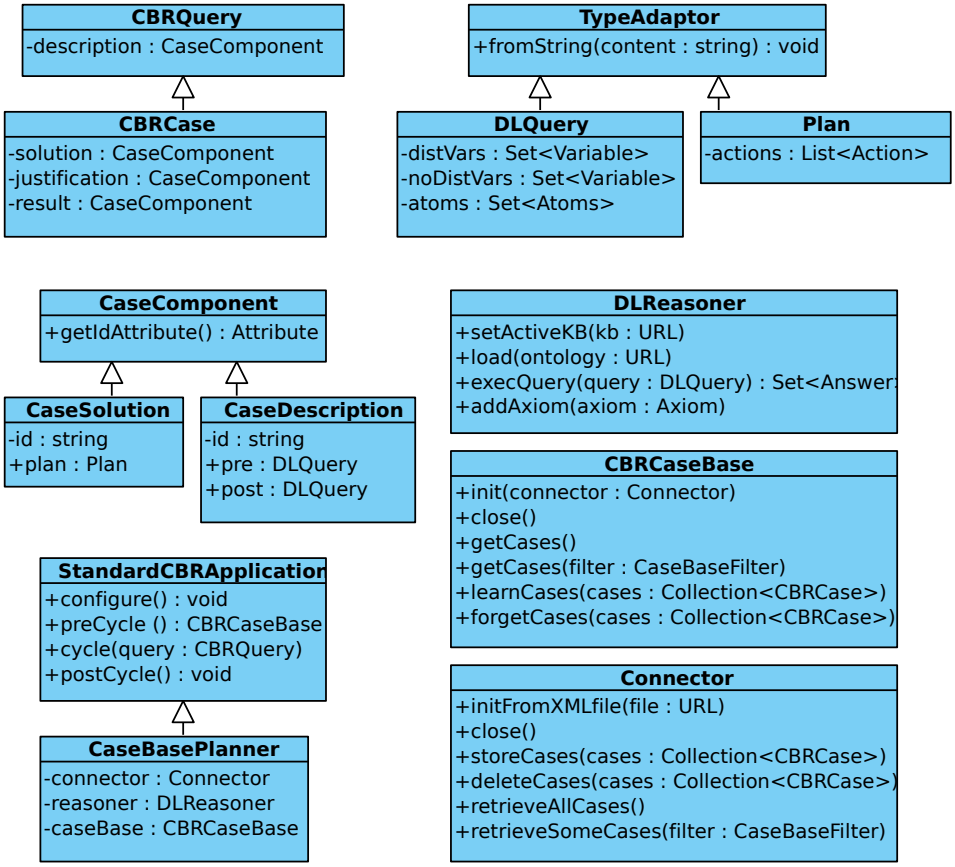


Figura 6.4: Integración en jCOLIBRI

ficheros de texto y posteriormente cargarlos en el razonador, o almacenarlos directamente como instancias en ficheros OWL-DL. jCOLIBRI proporciona conectores para trabajar con cualquiera de estos sistemas de persistencia. La configuración del conector elegido se realiza a través de un fichero XML en el que se especifican todos los parámetros necesarios.

Acceso al razonador

La biblioteca *OntoBridge* que proporciona jCOLIBRI para trabajar con razonadores OWL-DL es demasiado sencilla para nuestro sistema. *OntoBridge* no permite realizar consultas sobre una base de conocimiento y esta es una operación básica para nuestro planificador. El acceso al razonador tendríamos que hacerlo usando otro módulo más potente, seguramente basado en Jena.

Ciclo CBR

La clase *StandardCBRApplication* define 4 etapas para un sistema CBR: una configuración inicial, el pre-ciclo, el ciclo CBR y el post-ciclo. Durante el pre-ciclo cargaremos la base de casos en el razonador usando el conector adecuado y, de manera simétrica, durante el post-ciclo salvaremos la base de conocimiento a disco. También usaremos el post-ciclo para generalizar los pares problema - solución que hayamos resuelto con el planificador generativo, ya que esta es una operación costosa que no queremos realizar en cada iteración del ciclo CBR. Para implementar el ciclo CBR tenemos libertad absoluta, pero lo habitual es utiliza el modelo que divide el ciclo en 4 fases: recuperación, adaptación, revisión y apredizaje.

En nuestro planificador la fase de revisión consiste en: (1) recuperar los casos con una postcondición compatible con el objetivo del problema actual, y (2) comprobar cuales de estos casos son ejecutable en el estado inicial. Para resolver ambas operaciones necesitamos realizar consultas sobre la base de conocimiento usando la capa de acceso al razonador que hayamos definido.

Si todo va bien, tras la fase de revisión conseguiremos al menos un caso y una sustitución de variables que permite especializar el caso y adaptarlo al contexto del nuevo problema. Precisamente esto es lo que debemos hacer durante la fase de adaptación, aplicar la sustitución al caso para obtener un plan solución que sea ejecutable.

Nuestro planificador no tiene fase de revisión ya que las operaciones de generalización y especialización del caso se basan en operaciones lógicas y asumimos que el modelo del dominio es correcto. Si conseguimos un plan solución, debería ser válido.

La fase de aprendizaje en nuestro sistema se basa en un proceso de generalización de casos costoso. Por este motivo, en la fase de aprendizaje sólo almacenaremos los problemas resueltos por el planificador generativo para generalizarlos más tarde e incorporarlos a la base de casos en el post-ciclo.

6.4. Conclusiones

El principal problema de la planificación generativa basada en DLs está la complejidad computacional del problema. Trabajar con información incompleta y con teorías del dominio complejas aumenta la expresividad del lenguaje, pero también aumenta la dificultad de los procesos de razonamiento y el potencial espacio de búsqueda. Por este motivo, resulta interesante tratar de re-aprovechar los planes generados cuando necesitamos resolver problemas similares. Afortunadamente, la capacidad de trabajar con información incompleta e interpretar el conocimiento usando mundo abierto hace que los planes sean aplicables en muchas situaciones. Esto nos lleva a pensar que las aproximaciones CBR pueden resultar útiles.

En este capítulo hemos descrito una aproximación que utiliza el conocimiento representando en la ontología del dominio y las capacidades de inferencia de un razonador para: (1) generalizar el estado inicial de un problema para crear casos más aplicables, (2) recuperar y adaptar los casos relevantes al nuevo problema comparando los objetivos y condiciones iniciales. La limitación más importante de nuestra propuesta es que sólo somos capaces de reutilizar los planes si no es necesario modificar su estructura.

También hemos descrito el armazón jCOLIBRI, que facilita la creación de sistemas CBR, y hemos proporcionado algunas directrices que permitirían desarrollar el planificador basado en casos que describimos en el capítulo usando este armazón.

Capítulo 7

De la teoría a la práctica

En los capítulos anteriores hemos tratado distintos aspectos relacionados con el uso de ontologías para modelar el conocimiento estático asociado a los dominios de planificación. Todos los ejemplos que hemos planteado hasta ahora utilizan el dominio de la pizzería y consisten en combinar distintos ingredientes para conseguir pizzas con ciertas características. Este dominio es suficientemente sencillo para plantear ejemplos intuitivos y, al mismo tiempo, utiliza un vocabulario complejo que podemos describir de manera intuitiva usando una ontología. Sin embargo, el dominio de la pizzería no deja de ser un dominio de juguete sin interés práctico.

En este capítulo vamos a mostrar dos ejemplos en los que usamos DLPlan para resolver problemas en dominios más interesantes: la personalización de ejercicios en un videojuego educativo, y la definición del comportamiento de los personajes no jugadores (NPCs, del inglés *Non Player Characters*) en videojuegos. Ambos problemas se enmarcan en dominios con una estructura compleja donde resulta natural usar una representación basada en ontologías. Los problemas de planificación que vamos a resolver en estos dominios basan su dificultad, no tanto por el tamaño del espacio de búsqueda, como en la gestión del conocimiento asociado.

Comenzaremos el capítulo describiendo el videojuego educativo JV^{2}M [49, 51], que trata de enseñar cómo funciona un compilador de Java. En este sistema surge un problema común a muchos sistemas educativos: seleccionar adecuadamente el siguiente ejercicio que se debe proponer a un estudiante a partir de los datos almacenados en su perfil. En nuestro caso concreto disponemos de una base de ejercicios creada por expertos y el mejor ejercicio se selecciona usando una aproximación CBR. Sin embargo, puede ocurrir que ninguno de los ejercicios disponibles sea adecuado para los conocimientos actuales del estudiante. Llegados a este punto la aproximación CBR sugiere que tratemos de adaptar el ejercicio recuperado, que será el ejercicio disponible más similar al que buscamos. Nosotros proponemos el uso de planificación para realizar esta adaptación, utilizando para ello la ontología que describe

el conocimiento del dominio [131].

En la segunda parte de este capítulo veremos cómo podemos combinar la arquitectura basada en componentes que se utiliza en algunos videojuegos modernos con una descripción declarativa del dominio usando ontologías. Esta combinación nos permitirá definir un dominio de planificación con el que podremos ayudar a los diseñadores de un videojuego a definir el comportamiento de los NPCs. En concreto, describimos un proceso iterativo en el que ayudamos a construir las distintas ramas de los árboles de comportamiento [109, 110] que dirigirán a los NPCs durante el juego.

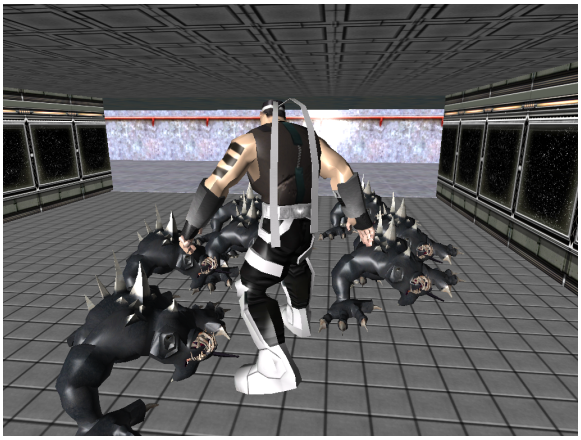
7.1. Adaptación de ejercicios en un videojuego educativo

El proceso de autoría de contenidos para videojuegos educativos es un proceso complejo y costoso. Muchos de estos sistemas utilizan un *corpus* de ejercicios creado por expertos educadores, entre los que se selecciona el ejercicio más adecuado para cada estudiante según su perfil. Aunque se debe intentar construir un corpus de ejercicios lo más completo posible, es inevitable llegar a situaciones en las que ninguno de los ejercicios disponibles se adapta bien a las necesidades del estudiante. En estos casos sería deseable disponer de algún mecanismo de adaptación automática que fuera capaz de modificar alguno de los ejercicios del corpus para adecuarlo al tipo de ejercicio que se busca.

Nosotros vamos a utilizar como dominio de pruebas JV^{2}M , un videojuego educativo desarrollado en nuestro grupo de investigación que enseña el funcionamiento de un compilador de Java. Este sistema utiliza un corpus de ejercicios creado por expertos, en este caso fragmentos de código Java, y elige el mejor ejercicio para cada estudiante usando una aproximación CBR. Todo el conocimiento del dominio está representado en una ontología que describe conceptos relacionados con el lenguaje Java, con el proceso de compilación y con el lenguaje de la máquina virtual que ejecuta los programas. Nosotros propondremos aplicar técnicas de planificación, usando el conocimiento representado en la ontología, para resolver la fase de adaptación del ciclo CBR y, de ese modo, aumentar la cobertura de cada ejercicio. Como veremos más tarde, en este dominio resulta sencillo usar esta aproximación porque el proceso de adaptación se reduce a realizar cambios sintácticos en la estructura de un programa.

7.1.1. Javy: juega y aprende a compilar Java

JV^{2}M [49, 51] o *Javy 2*, como nos gusta referirnos a él en el grupo de investigación, es un videojuego educativo que trata de enseñar a los estudiantes el funcionamiento de un compilador. Cada nivel del juego consiste en un



(a) Buscando objetos por el mapa

(b) Escribiendo código en una terminal

Figura 7.1: Aprende a compilar Java jugando con JV^{2}M .

pequeño programa Java que el alumno debe compilar, transformándolo en secuencias de instrucciones del lenguaje ensamblador de la Máquina Virtual de Java (JVM). *Javy* sigue una aproximación a la enseñanza conocida como *learning-by-doing*, en la que los estudiantes aprenden los conceptos implicados en una materia a medida que resuelven los problemas o retos que se les plantean.

Cualquier videojuego educativo debe combinar la componente educativa (lo que se desea enseñar) con una componente de tipo lúdico que "enganche" al estudiante durante el juego. Aunque aprender a compilar un programa puede parecer una tarea tediosa, los creadores de *Javy* han ideado un conjunto de dinámicas que transforman el problema en algo divertido [50]. En el juego, el estudiante se ve sumergido en un mundo 3D donde su avatar debe recoger ciertos objetos necesarios para escribir más tarde las instrucciones de

la JVM en una terminal (figura 7.1b). Los objetos se encuentran repartidos por el mapa, por lo que el estudiante debe explorar distintas regiones eliminando los enemigos que encuentre a su paso (figura 7.1a). Además, según va escribiendo la secuencia de instrucciones correcta en la terminal, el mundo a su alrededor va cambiando para reflejar la ejecución de dichas instrucciones en la JVM. Por último, con idea de aumentar la componente adictiva del juego, *Javy* está evolucionando hacia un juego multijugador en el que distintos estudiantes compiten por completar los ejercicios primero, a la vez que intentan evitar que los demás jugadores completen los suyos.

El sistema debe supervisar al estudiante durante la partida para poder ofrecerle ciertas ayudas cuando se atasca o comete errores. *Javy* conoce la solución correcta de cada ejercicio y sabe cómo compararla con la solución que el estudiante está construyendo. Cuando el sistema detecta que el estudiante ha cometido un error tiene dos opciones: interrumpirle y mostrarle el error, o dejar que siga adelante esperando que se de cuenta del error por sí mismo. En este sentido hay que llegar a un punto de equilibrio, el jugador debe sentir que es libre para explorar distintas opciones (algunas incorrectas) pero debemos evitar que llegue a sentirse frustrado por no encontrar la solución.

Para evitar llegar a este tipo de situaciones, los ejercicios se deben elegir en orden creciente de dificultad. Antes de cada nivel o *episodio de aprendizaje*, el *módulo pedagógico* del sistema analiza la información almacenada en el *perfil del estudiante* para seleccionar los conceptos teóricos que debe practicar a continuación. También es importante marcar aquellos conceptos que, debido a su dificultad, aún no deben aparecer. Después, usando toda esta información, el sistema propone al estudiante un ejercicio, en este caso un programa Java que pone en práctica los conceptos teóricos seleccionados. En nuestro caso los conceptos teóricos que se enseñan están muy ligados al mundo de la programación: *StaticMethod*, *AddExpression*, *WhileStatement*, etc. De este modo, los primeros ejercicios a los que se enfrenta un estudiante consisten en compilar sencillas expresiones aritméticas, pero a medida que juegue y aprenda tendrá que enfrentarse a problemas más complejos.

En principio, en este dominio sería posible crear los ejercicios automáticamente usando las estructuras de código que se necesitan en cada momento. Sin embargo, los ejercicios creados de esta forma, aunque correctos desde el punto de sintáctico, no representarían programas interesantes. Desde el punto de vista del alumno motiva más aprender a compilar programas "reales" que trabajar sobre trozos de código generados de manera aleatoria.

Por este motivo, en *Javy* se ha optado por usar una aproximación basada en casos, donde expertos pedagogos deben crear de manera manual un *corpus de ejercicios*. Los ejercicios se anotan con etiquetas semánticas que describen los conceptos del dominio que abarca. Este marcado semántico se utiliza durante la fase de recuperación para seleccionar el ejercicio que se adecúa

```
public static int euclides(int a, int b) {
  int res;
  if ( (a <= 0) || (b <= 0) )
     res = 0;
  else {
     while (a != b) {
        if (a < b)
           b -= a;
        else
           a -= b;
     }
     res = a;
  }
  return res;
}
```

Figura 7.2: Algoritmo de Euclides (M.C.D.)

más a los requisitos seleccionados por el módulo pedagógico. Por ejemplo, el algoritmo de Euclides que se muestra en la figura 7.2, y que calcula el máximo común divisor de dos números, se marcaría con etiquetas semánticas del tipo *IfStatement*, *WhileStatement* y *MinusExpression* entre otras.

Desafortunadamente, construir un corpus de ejercicios lo suficientemente completo para este tipo de sistemas implica un gran esfuerzo de creación de contenidos. Los programas educativos necesitan disponer de un gran número de ejercicios para evitar repeticiones y crear la ilusión de una fuente interminable de problemas. Surge, por tanto, la necesidad de encontrar un equilibrio entre creación manual y creación automática de ejercicios. En este contexto debemos prestar especial atención a la fase de adaptación del sistema CBR, y aprovechar la representación rica en conocimiento del dominio para personalizar los ejercicios recuperados.

7.1.2. Adaptación en sistemas CBR usando planificación

La fase de adaptación en los sistemas CBR es, por lo general, mucho más compleja de definir que la fase de recuperación. De hecho la mayor parte de los sistemas CBR no adaptan los casos recuperados o lo hacen usando reglas sencillas creadas a medida en cada aplicación. La adaptación es una tarea compleja que requiere una buena comprensión del dominio, e incluso los expertos suelen tener problemas para describir lo que la experiencia les ha enseñado. Sin embargo, los sistemas capaces de realizar adaptaciones complejas tienen la ventaja de que no requieren tantos casos en la base de conocimiento, ya que la cobertura de cada caso almacenado aumenta.

La figura 7.3 muestra cómo funciona un sistema CBR con fase de adap-

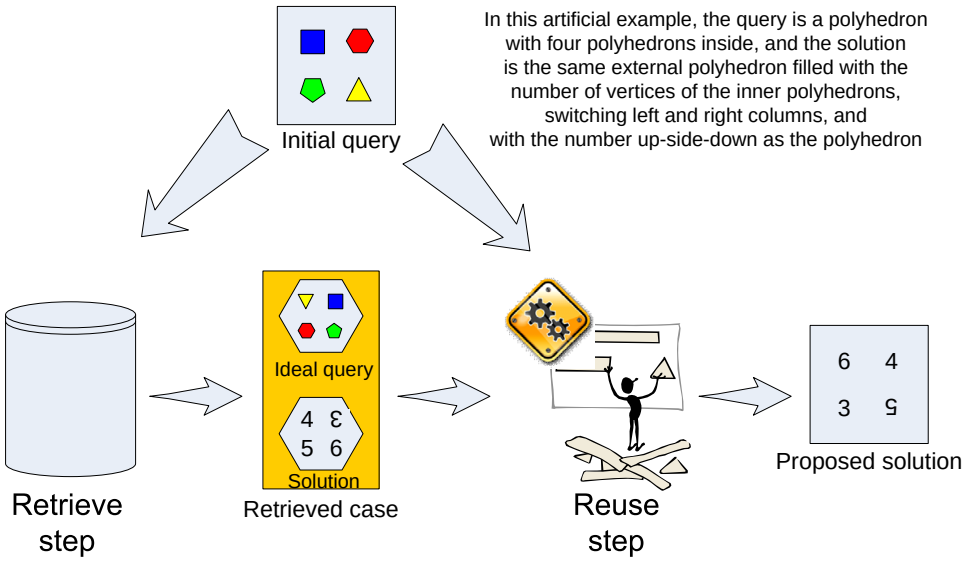


Figura 7.3: Adaptación en un sistema CBR.

tación. El sistema recibe una consulta inicial que describe el problema que tratamos de resolver. Durante la fase de recuperación buscamos en la base de casos episodios pasados (pares problema - solución) con un problema similar al actual. Por lo general, el problema del caso recuperado no coincidirá con el problema actual por lo que la solución puede no ser directamente aplicable al nuevo problema. Precisamente ese es el objetivo de la fase de adaptación: modificar la solución del caso recuperado para convertirla en una solución válida del problema actual. Y para adaptar la solución, lo lógico es tener en cuenta las diferencias entre el problema actual y el problema recuperado.

Podemos plantear la fase de adaptación de un sistema CBR como un problema de planificación usando las siguientes ideas:

- El estado inicial del problema describe el caso recuperado.
- Los objetivos del problema se describen usando la consulta inicial del sistema CBR.
- Necesitamos definir operadores que transformen el estado inicial (caso recuperado) hasta alcanzar un nuevo estado (caso) que satisfaga los objetivos del problema (consulta inicial del sistema CBR).

La figura 7.4 muestra un ejemplo. En la parte izquierda de la imagen está el caso recuperado, que se compone de dos partes: la descripción del problema y la descripción de la solución. En la parte superior derecha de la imagen está la descripción del problema que se pretende resolver. El problema de

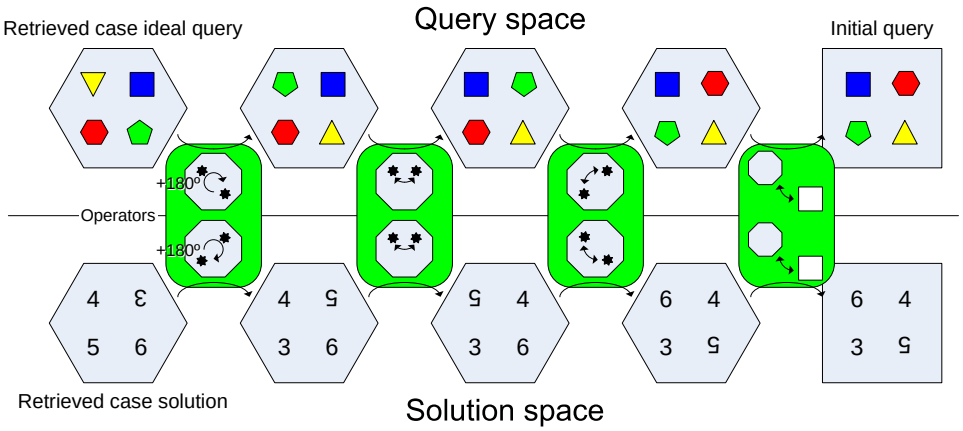


Figura 7.4: Búsqueda en el espacio de problemas.

planificación consiste en encontrar la secuencia de operadores que transforma el caso recuperado hasta alcanzar otro caso que contenga el problema que pretendemos resolver. En la imagen los operadores rotan e intercambian las figuras geométricas internas o modifican el polígono externo.

Si planteamos el problema de esta manera, los operadores de planificación no sólo deben modificar la descripción del problema, también deben transformar la solución de manera consistente, para que cada paso siga representando una pareja válida problema - solución. También es interesante resaltar que el planificador elige los operadores teniendo en cuenta *sólo* las descripciones del problema actual y del problema objetivo, y el espacio de soluciones se recorre como efecto colateral al recorrer el espacio de problemas. Llamaremos a esta aproximación *búsqueda en el espacio de problemas*.

Definir este tipo de operadores *duales*, que modifican tanto la descripción del problema como su solución, puede ser muy difícil o incluso imposible en ciertos dominios. Necesitaríamos comprender las relaciones entre los problemas y sus soluciones, o al menos entender cómo afectan los cambios atómicos que realizamos en las descripciones de los problemas a sus soluciones.

La imagen 7.5 muestra otro enfoque distinto al que llamaremos *planificación en el espacio de soluciones*. En este caso los operadores sólo transforman la solución del caso recuperado, sin tener en cuenta cómo afectan estos cambios a la descripción del problema. También necesitamos una función que, dados la descripción de un problema y una solución candidata, compruebe si la solución propuesta es válida. El proceso de búsqueda continúa transformando la solución del caso recuperado hasta alcanzar una solución que resuelve el problema actual. En algunos dominios puede resultar más sencillo escribir este tipo de operadores que transforman soluciones que los operadores duales de la aproximación anterior.

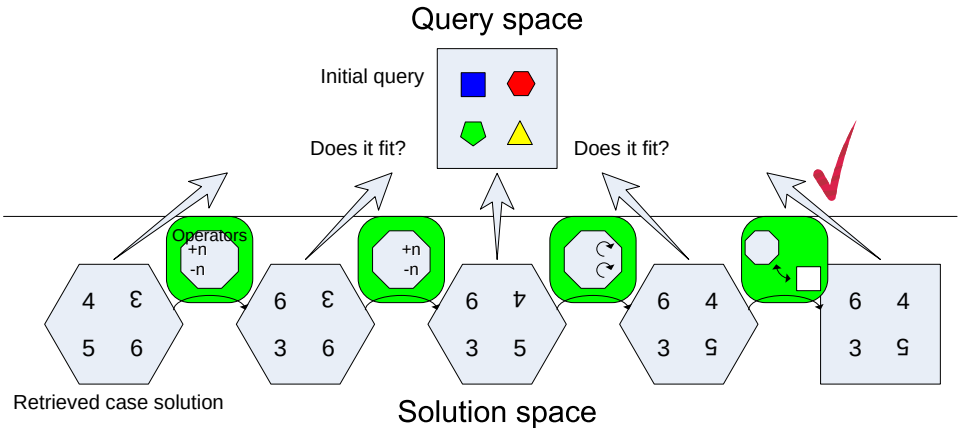


Figura 7.5: Búsqueda en el espacio de soluciones.

Definir buenas heurísticas que guíen la búsqueda en el espacio de soluciones puede ser problemático, ya que sólo disponemos de una función que comprueba si un estado es un estado objetivo. Sin embargo, debemos recordar que estamos trabajando con la solución del caso recuperado, que contiene un problema similar al actual. Por tanto, es de esperar que la solución recuperada esté *cerca* de la solución buscada y el plan que produce la adaptación no sea muy largo. Al fin y al cabo, toda la filosofía CBR se sustenta en la hipótesis de que *problemas similares suelen tener soluciones similares.*

En cualquier caso, utilizar técnicas de planificación para adaptar casos en lugar de mecanismos *ad hoc* tiene ventajas evidentes:

- *Los operadores representan cambios atómicos fáciles de escribir.* La tarea del planificador consiste en combinar los operadores para crear adaptaciones complejas.
- *El conocimiento de adaptación se representa usando un lenguaje lógico estándar*, mediante precondiciones y efectos. Este enfoque declarativo facilita que el conocimiento de adaptación se comparta y reutilice.
- *Los planificadores son buenos resolviendo este tipo de problemas*, y saben cómo recorrer el espacio de búsqueda de manera inteligente.

No sería honrado terminar esta sección sin hace la siguiente reflexión. Uno de los puntos fuertes de los sistemas CBR es que los casos pueden contener conocimiento implícito sobre el dominio que no somos capaces de representar de manera explícita. Una pregunta que nos debemos plantear es si durante el proceso de planificación, después de aplicar varios operadores, los casos a los que llegamos siguen siendo correctos. La respuesta a esta pregunta queda fuera del alcance de este trabajo, pero es obvio que los operadores de

planificación deben elegirse con mucho cuidado, y existe un equilibrio entre capacidad de adaptación y preservación de la corrección de los casos.

7.1.3. Representación de ejercicios en Javy

Los sistemas educativos requieren habitualmente mucha información sobre el dominio. El sistema no sólo debe proponer ejercicios interesantes al alumno, también debe observar su evolución durante el juego para ayudarle cuando sea preciso. *Javy* no es una excepción, el dominio de este sistema abarca conceptos sobre el lenguaje de programación Java, sobre la máquina virtual y sobre el proceso de compilación en general. Para representar toda esta información se decidió modelar el conocimiento del dominio usando ontologías OWL-DL.

El uso de ontologías OWL-DL para representar el conocimiento del dominio en *Javy* resulta interesante por varios motivos. En primer lugar, disponemos que intuitivas herramientas de edición que facilitan el proceso de creación y almacenan el conocimiento de manera declarativa usando un lenguaje estándar. En segundo lugar, la expresividad de las DLs facilita la definición de un vocabulario bastante completo que utilizaremos como etiquetas semánticas para marcar los ejercicios. Además, durante la fase de recuperación del ciclo CBR podemos valernos de medidas de similitud que explotan la estructura del dominio descrita en la ontología para recuperar ejercicios interesantes para el estudiante. Finalmente, las capacidades de razonamiento de las DLs nos permitirán detectar inconsistencias de la base de conocimiento y, de ese modo, detectar posibles errores durante el proceso de autoría de ejercicios.

La figura 7.6 muestra una pequeña parte de la ontología que usamos para describir el dominio. Los casos son pequeños programas Java que se representan mediante grafos de instancias definidos usando el vocabulario de la ontología. Por ejemplo, la figura 7.7 muestra el grafo que resulta al formalizar en DLs un trozo de código sencillo. En este caso, la instrucción *if* se representa usando un individuo del tipo *IfStatement*, que se relaciona mediante la propiedad *hasCondition* con otro individuo de tipo *BooleanExpression* (una "o" lógica), y mediante la propiedad *hasYesBranch* con el cuerpo del *if* (la asignación).

No debemos confundir el formalismo que utilizamos para representar los ejercicios con la herramienta que los educadores usan durante el proceso de autoría. Los ejercicios se introducen en el sistema como programas Java escritos en ficheros de texto y, posteriormente, se traducen de manera automática a un grafo de instancias equivalente.

Finalmente, nos gustaría aclarar que la decisión de usar OWL-DL para representar la información en *Javy* se tomó antes de empezar a trabajar en DLPlan. Es decir, las DLs son una manera natural de describir este tipo

Figura 7.6: Algunos conceptos de la ontología que usamos en *Javy*.

de dominios, y disponer de un planificador que use el mismo formalismo nos brinda la oportunidad de utilizar planificación en estos sistemas sin necesidad de traducir la base de conocimiento.

7.1.4. Ejemplo de adaptación

Cuando *Javy* tiene que proponer un nuevo ejercicio a un estudiante lo primero que hace es decidir qué tipo de ejercicio es el más adecuado. Para ello consulta el perfil del estudiante y selecciona tanto los conceptos que se pretenden enseñar o reforzar, como los conceptos que aún no deben aparecer en el siguiente ejercicio. Usando esta información el sistema construye una consulta y, usando medidas de similitud basadas en la ontología del dominio, se recupera el ejercicio más similar de la base de casos. Si el caso recuperado encaja con la consulta, podemos usarlo directamente como siguiente ejercicio.

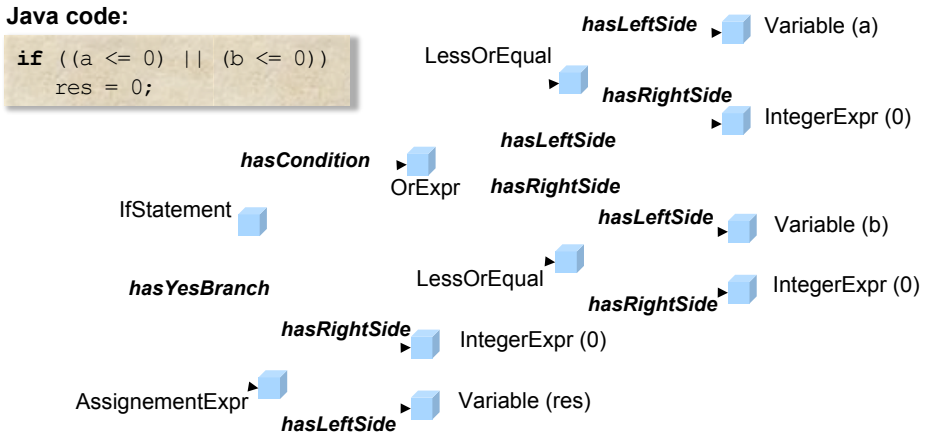


Figura 7.7: Cómo se representa el código Java en la ontología.

En caso contrario, debemos adaptar el ejercicio recuperado hasta que encaje con los requisitos establecidos en la consulta.

Supongamos que, analizando el perfil de un estudiante, el sistema decide que el siguiente ejercicio debe contener instrucciones *if* y *while*, pero no deben aparecer los operadores lógicos *OR* (//) ni *NOT* (*!*). Esta situación se puede dar si, por ejemplo, el estudiante aún no sabe cómo compilar algunas expresiones de tipo lógico, pero ya ha practicado las instrucciones *if* y *while* por separado y ahora queremos retarle con un ejercicio que las combina.

Supongamos también que, usando la consulta anterior, el ejercicio de la base de casos más similar es el algoritmo de Euclides que ya mostramos en la figura 7.2. Este programa efectivamente combina sentencias *if* y *while*, pero también contiene la expresión lógica *OR* que pretendíamos evitar. En este caso debemos adaptar el ejercicio recuperado, y para hacerlo usaremos operadores de planificación que realizan cambios sintácticos en la estructura de programa sin cambiar su semántica.

Los operadores necesarios para resolver este ejemplo se muestran en la figura 7.8. El primer operador (*swapIfBranches*) intercambia las dos ramas de una instrucción *if-then-else* negando la condición booleana e intercambiando las sentencias de las partes *then* y *else*. El segundo operador simplifica una expresión lógica usando la propiedad $\neg(a \leq b) \Rightarrow a > b$. De manera similar podemos definir operadores para simplificar $\neg(a = b) \Rightarrow a \neq b$, $\neg(a < b) \Rightarrow a \geq b$, etc. Finalmente definimos operadores que representen las leyes de De Morgan: $\neg(a \vee b) \Rightarrow \neg a \wedge \neg b$, $\neg(a \wedge b) \Rightarrow \neg a \vee \neg b$.

La figura 7.9 muestra cómo podemos utilizar los operadores anteriores para adaptar el algoritmo de Euclides y eliminar la molesta expresión *OR*. En primer lugar usamos el operador *swapIfBranches* sobre el *if* más externo. A continuación una de las leyes de De Morgan para eliminar la *OR* negada.

oper swapIfBranches
vars: ?if1 , ?c1, ?br1, ?br2, ?c2
pre: IfInstruction (?if1), hasCondition(?if1, ?c1), hasIfYesBranch(?if1, ?br1),
hasIfNoBranch(?if1, ?br2), NewItem(?c2)
del: hasCondition(?if1, ?c1), hasYesBranch(?if1, ?br1),
hasNoBranch(?if1, ?br2), NewItem(?c2)
add: NotExpr(?c2), hasSubExpr(?c2, ?c1), hasCondition(?if1, ?c2),
hasYesBranch(?if1, ?br2), hasNoBranch(?if1, ?br1)

oper notLessOrEqual
vars: ?e1, ?e2, ?le , ?re
pre: NotExpr(?e1), hasSubExpr(?e1, ?e2), LessOrEqualThanExpr(?e2),
hasSubexpr(?e2, ?le), hasSubexpr2(?e2, ?re)
del: NotExpr(?e1), hasSubExpr(?e1, ?e2), hasSubexpr(?e2, ?le),
hasSubexpr2(?e2, ?re)
add: GreaterThanExpr(?e1), hasSubexpr(?e1, ?le), hasSubexpr2(?e1, ?re),
NewItem(?e2)

oper notOr
vars: ?e1, ?e2, ?le , ?re , ?n1, ?n2
pre: NotExpr(?e1), hasSubExpr(?e1, ?e2), OrExpr(?e2), hasSubexpr(?e2, ?le),
hasSubexpr2(?e2, ?re), NewItem(?n1), NewItem(?n2)
del: NotExpr(?e1), hasSubExpr(?e1, ?e2), hasSubexpr(?e2, ?le),
hasSubexpr2(?e2, ?re), NewItem(?n1), NewItem(?n2)
add: AndExpr(?e1), hasSubexpr(?e1,?n1), hasSubexpr2(?e1,?n2), NotExpr(?n1),
NotExpr(?n2), hasSubexpr(?n1,?le), hasSubexpr(?n2,?re), NewItem(?e2)

Figura 7.8: Operadores de adaptación

Llegados a este punto el planificador ha conseguido eliminar el operador *OR* pero también ha añadido dos operadores *NOT* que, recordemos, no debían aparecer en el ejercicio. Sin embargo, podemos resolver fácilmente esta situación aplicando el operador que simplifica una negación lógica dos veces. Ahora sí, hemos conseguido un nuevo algoritmo de Euclides que cumple las condiciones requeridas (tiene *if* y *while* pero no *OR* ni *NOT*), y el caso ha sido adaptado con éxito.

7.1.5. Conclusiones

En este ejemplo hemos demostrado cómo podemos utilizar planificación en un videojuego educativo para adaptar los ejercicios disponibles cuando ninguno de ellos es adecuado para un estudiante. En realidad esto no es sino un ejemplo concreto de una aproximación más general: usar planificación para resolver la fase de adaptación de un sistema CBR, aumentando la aplicabilidad de cada caso y reduciendo, por tanto, el número de casos necesarios. La principal ventaja de esta aproximación es que sólo necesitamos describir

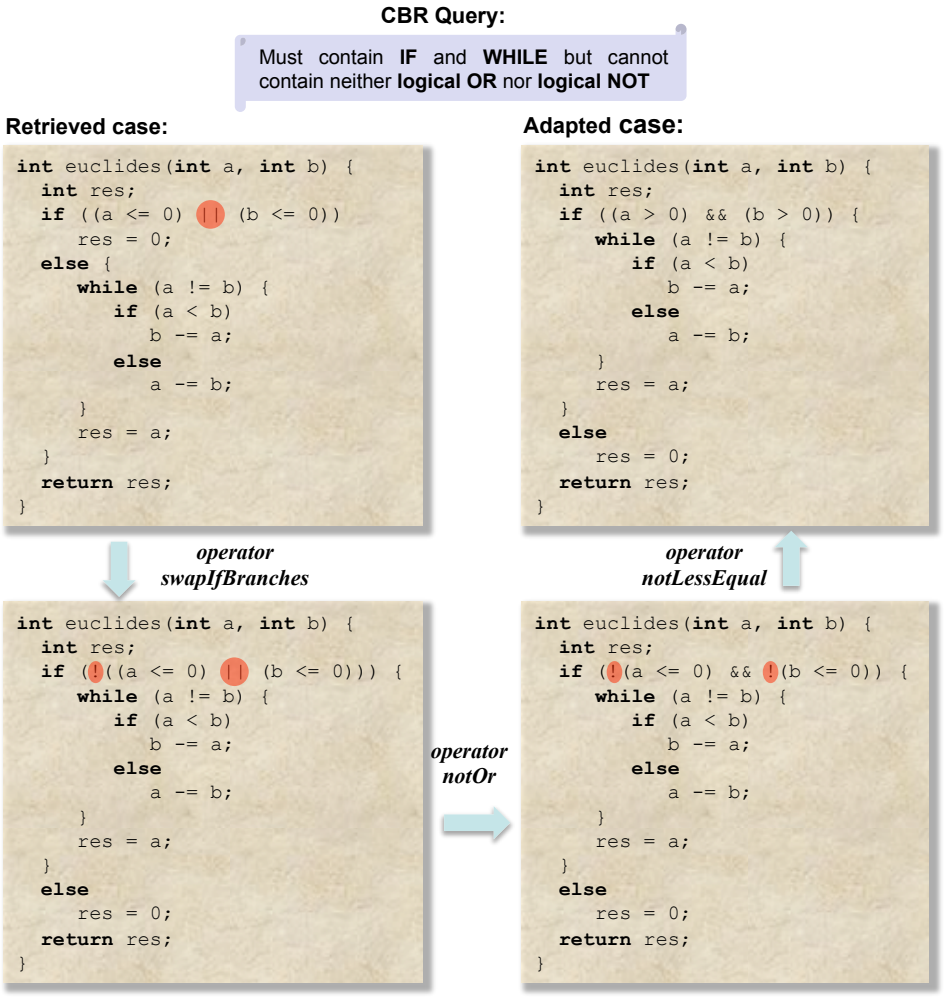


Figura 7.9: Adaptación del algoritmo de Euclides.

pequeñas transformaciones atómicas como operadores y el planificador se encargará de combinarlas para realizar adaptaciones complejas. El principal problema está en la dificultad que entraña adquirir y modelar el conocimiento de adaptación para garantizar que los planes hacen transformaciones válidas sobre los casos recuperados.

En concreto hemos usado el videojuego educativo JV^{2}M , que enseña conceptos relacionados con la compilación de programas Java. El sistema cuenta con corpus de ejercicios creado por expertos, y los ejercicios están anotados con etiquetas semánticas usando el vocabulario de una ontología del dominio formalizada en OWL-DL. Usando el conocimiento de la ontología hemos transformado la adaptación de ejercicios en un problema de planificación en

el que los operadores realizan cambios sintácticos en los programas sin afectar a su semántica. Esto nos garantiza que los ejercicios adaptados siguen siendo ejercicios válidos para el sistema y tienen "sentido" para el estudiante. De las dos aproximaciones teóricas que hemos descrito, nosotros hemos usado la de búsqueda en el espacio de soluciones, ya que los operadores sólo afectan a los programa Java. En JV^{2}M la descripción de los casos contiene las etiquetas semánticas que describen los programas, y la solución de los casos son los programas Java en sí mismos.

7.2. Ayuda al diseño de NPCs en videojuegos

Basta consular las últimas ediciones de la conocida serie *AI Game Programming Wisdom* [109, 110] para comprobar que los árboles de comportamiento (BTs, del inglés *Behavior Trees*) son una tecnología incipiente pero que cuenta con bastante respaldo a la hora definir el comportamiento de los personajes no jugadores (NPCs) en los videojuegos modernos. Los BTs surgen como evolución de las máquinas de estados finitos y tratan de resolver sus problemas de escalabilidad haciendo especial énfasis en la reutilización de comportamientos.

Los BTs son una potente herramienta que permite a los diseñadores definir complejos comportamientos con los que crear la ilusión de personajes inteligentes que reaccionan de manera coherente a lo que pasa en su entorno. Sin embargo, la creación de BTs sigue siendo un proceso complejo, sobre todo para los diseñadores, que son los responsables de definir el comportamiento de los NPCs, porque suelen carecer de conocimientos de programación [69, 70]. Para aliviar este problema, las compañías de videojuegos tienden a construir herramientas visuales que permiten definir estos árboles eligiendo entre un conjunto predefinido de nodos, condiciones y acciones básicas. Pero incluso con estas herramientas, en la práctica sigue existiendo tensión entre la libertad que los diseñadores necesitan para crear la parte narrativa del juego y el esfuerzo que los programadores deben realizar para corregir la IA creada por no programadores. Nuestra propuesta consiste en ayudar a los diseñadores durante el proceso de autoría de comportamientos usando técnicas de planificación [133].

Un inconveniente de usar técnicas de IA que explotan una descripción declarativa del dominio es precisamente la necesidad de modelar el dominio usando el formalismo correspondiente. En este caso necesitamos modelar tanto las entidades del juego como las acciones básicas que los personajes pueden realizar. Para intentar minimizar la distancia que existe entre la IA académica y los videojuegos industriales, proponemos generar el dominio de planificación mediante una extensión de la aproximación basada en componentes que se usa habitualmente en la industria de los videojuegos para describir las entidades del mundo.

Comenzaremos esta segunda mitad del capítulo describiendo de forma breve las dos tecnologías que acabamos de introducir: los sistemas basados en componentes y los árboles de comportamiento. A continuación describiremos nuestra propuesta, en la que usamos planificación para ayudar a los diseñadores a crear los BTs, y veremos cómo podemos aprovechar la arquitectura de componentes para modelar el conocimiento del dominio. Finalmente, mostraremos un ejemplo concreto en el que asumiremos el papel de diseñador para construir un árbol de comportamiento.

7.2.1. Tecnología

7.2.1.1. Arquitectura basada en componentes

La capa responsable de la gestión de las distintas entidades que aparecen en un videojuego se suele crear usando un lenguaje orientado a objetivos, habitualmente C++. Una manera natural de modelar las entidades del mundo es utilizar una jerarquía de clases en la que partimos de una clase base común que vamos especializando hasta definir las entidades concretas del juego en las hojas de la jerarquía.

Sin embargo, el uso de herencia para modelar este tipo de jerarquías complejas provoca, entre otros problemas, un incremento en el tiempo de compilación [77], un código base difícil de entender y clases muy grandes en la parte alta de la jerarquía. Por ejemplo, la clase base de *Half-Life 1* tenía 87 métodos y 20 atributos públicos, y la de *Los Sims 1* más de 100 métodos.

Estos problemas han motivado el uso de otro tipo de arquitecturas basadas en *componentes* reutilizables [143, 119, 27, 43]. En lugar de crear una clase distinta para definir el comportamiento concreto de cada entidad, ahora las entidades se modelan como simples contenedores de componentes, y en cada componente se implementa una funcionalidad o habilidad básica. Desde el punto de vista del programador los componentes son clases que implementan un interfaz común *IComponent* y las entidades son listas de objetos *IComponent*.

Como todos los componentes tienen un interfaz común, independiente de su funcionalidad, la comunicación entre ellos se hace mediante paso de mensajes. Cada mensaje encapsula algún tipo de información, que para algunos componentes será útil y para otros no. Como las entidades no saben que tipo de componentes contienen deben hacer llegar los mensajes a todos ellos, siendo cada componente responsable de procesar o ignorar cada mensaje. Por ejemplo, cuando el componente de IA de una entidad, responsable de decidir su comportamiento, decide que la entidad debe llegar a cierto lugar enviará un mensaje con la petición. La entidad, a su vez, retransmite dicho mensaje a cada uno de sus componentes, hasta que el componente responsable del movimiento lo intercepta, calcula el camino a seguir y emite periódicamente mensajes de cambio de posición. Estos mensajes a su vez serán procesados

```
<blueprint>
<entity type="goblin" ontType="Goblin" parentOnt="Monster">
  <components list="Take, MoveTo, TakeCover, MeleeAttack,
                    LongRangeAttack, ChargeAt, ..."/>
  <attributes>
    <attrib name = "strength" value = "weak"/>
    <attrib name = "weapon-tech"
            value = "rudimentary, elaborate"/>
    <attrib name = "height" value = "short"/>
    ...
  </attributes>
</entity>
...
</blueprints>
```

Figura 7.10: Fragmento del fichero *blueprints* que define las entidades.

por el componente encargado de almacenar la posición de la entidad.

En los juegos suelen existir un gran número de entidades distintas (puertas, armas, pociones, monstruos, ...), así que resulta útil describirlas en ficheros de texto y usar una arquitectura dirigida por datos. Este tipo de ficheros se denominan *blueprints* y tienen un aspecto similar al de la figura 7.10. Usando esta aproximación, el motor del juego es independiente de los tipos de entidades, lo que facilita la creación y posterior gestión de objetos y personajes.

La arquitectura de componentes se adapta de manera natural a este planteamiento. Para describir cada tipo de entidad sólo necesitamos enumerar sus componentes, que implementan las distintas habilidades de la entidad, y sus atributos, que se usan como parámetros de los componentes para ajustar su funcionamiento. Por ejemplo, la figura 7.10 describe una entidad de tipo *Goblin*. Los componentes de esta entidad determinan que un goblin es capaz de coger cosas (*Take*), desplazarse por el mapa (*MoveTo*), ocultarse detrás de objetos (*TakeCover*), atacar cuerpo a cuerpo y a distancia (*MeleeAttack* y *LongRangeAttack*), cargar contra un enemigo (*ChargeAt*), etc. Por otra parte, los atributos de la entidad goblin modifican el comportamiento final de los componentes. Por ejemplo, el atributo *strength* restringe el tipo de objetos que el componente *Take* puede coger, el atributo *weapon-tech* (tecnología de las armas) afecta al componente *MeleeAttack*, y el atributo *height* determina el tipo de objetos que el componente *TakeCover* considerará coberturas válidas.

7.2.1.2. Árboles de comportamiento

Los BTs tienen dos propiedades que los hacen especialmente atractivos para definir el comportamiento de NPCs: (1) permiten definir un modelo de comportamiento dirigido por objetivos, y (2) permiten trabajar de manera incremental y fomentan la reusabilidad, ya que los BTs se pueden combinar para crear otros BTs que describan comportamientos cada vez más complejos.

La figura 7.16 (ver más adelante) muestra un ejemplo concreto de BT en el que se describen distintas formas en las que un NPC puede intentar conseguir un objeto. Los nodos internos del árbol representan comportamientos compuestos, y se descomponen en otros comportamientos más sencillos hasta alcanzar las hojas del árbol que corresponden a acciones concretas que el NPC puede ejecutar. Cada nodo del árbol se representa mediante un comportamiento (compuesto o acción primitiva), un conjunto de parámetros que en el momento de ejecución se vincularán a entidades reales del mundo, y un conjunto de condiciones o guardas que determinan si el nodo se puede activar.

Se han propuesto mucho tipos de comportamiento compuestos (nodos internos) en la literatura, pero nosotros nos centraremos en los 2 más sencillos: *secuencias* y *selectores*. Un nodo de tipo secuencia intenta ejecutar sus hijos en el mismo orden en el que se definen, y sólo se considera que el comportamiento ha terminado con éxito si todos los hijos se ejecutan de manera exitosa. Los selectores, sin embargo, representan distintas formas de alcanzar un objetivo, por lo que se completan con éxito si cualquiera de sus hijos se ejecuta con éxito. Dentro de los selectores existen distintas tipos de nodos en función de cómo se determina el hijo que se ejecuta cuando hay varios candidatos (prioridad estática, prioridad dinámica, aleatorio). Finalmente existen otros tipos de nodos, que no veremos, para ejecutar varios comportamientos de manera simultánea, decoradores, etc.

También existen distintos modelos de ejecución de BTs, pero nosotros vamos a utilizar uno de los más sencillos donde sólo existe una rama activa del árbol en cada ocasión que va desde la raíz hasta una hoja. En cada ciclo de actualización del juego se volverán a evaluar algunas de las condiciones del árbol y algunas acciones finalizarán (con éxito o error), lo que puede producir que cambie la rama activa del árbol y, por tanto, el comportamiento del NPC.

Desde el punto de vista de la creación de BTs existen dos actores implicados: los programadores y los diseñadores. Los programadores deben crear un repositorio de acciones básicas potentes pero versátiles. Los diseñadores son los responsables de combinar estas acciones básicas usando los distintos tipos de nodos compuestos, y construir los árboles que guiarán el comportamiento de los NPC. Una ventaja de utilizar BTs es que los árboles que definen comportamientos sencillos se pueden utilizar como ramas de otros árboles para definir comportamientos más elaborados. También es importante entender que, aunque los diseñadores no suelen tener conocimiento de programación,

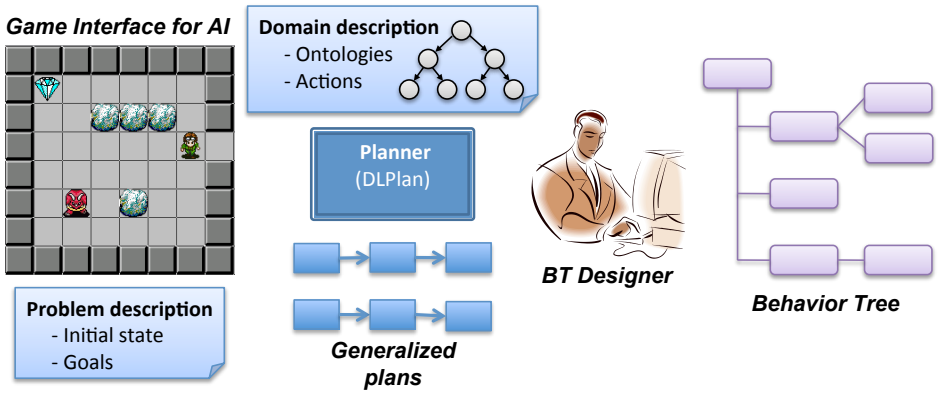


Figura 7.11: Proceso de creación de árboles de comportamiento.

tienen una mejor comprensión del juego a nivel de argumento, jugabilidad, niveles de dificultad, etc.

7.2.2. Planificación para facilitar la creación de BTs

La creación de BTs es una tarea compleja que se suele realizar mediante un costoso proceso de prueba y error. Debemos tener en cuenta que la colección de acciones básicas puede ser muy grande y variada, y que dichas acciones se pueden combinar de cientos o incluso miles de maneras distintas. Además, conseguir que los NPCs actúen de manera coherente en un entorno tan complejo como el que suele existir en los videojuegos modernos es realmente complicado, debido al inmenso número de situaciones e interacciones que pueden surgir. Finalmente, es necesario definir de manera precisa el comportamiento de las acciones básicas, ya que quienes las implementan (los programadores) y quienes las usan para definir comportamientos de alto nivel (los diseñadores) suelen ser distintos grupos de personas con distinto tipo de formación. La consecuencia de todo esto es que la calidad final de los BTs depende en gran medida de la pericia y experiencia de los diseñadores, lo que encarece todo el proceso de producción. Por este motivo, debemos desarrollar herramientas que faciliten lo máximo posible el proceso de autoría de comportamientos.

Intuitivamente parece razonable proponer el uso de técnicas de planificación para ayudar a los diseñadores en su tarea. Al fin y al cabo, lo que estamos haciendo es combinar un conjunto de acciones básicas tratando de alcanzar ciertos objetivos. Sin embargo, no debemos pasar por alto que para poder utilizar planificación necesitamos una representación formal del dominio y de las acciones disponibles. Formalizar todo este conocimiento requiere cierto esfuerzo extra, pero debemos considerarlo una inversión, ya que, previsiblemente, abarataremos los costes asociados a la autoría de comportamientos.

La figura 7.11 resume nuestra propuesta. En primer lugar, usando una interfaz gráfica, que puede ser una versión simplificada de la interfaz del juego, el diseñador establece un escenario particular y unos objetivos concretos para un NPC. A continuación, el sistema crea automáticamente una representación simbólica del estado del juego y de los objetivos propuestos, y el planificador calcula todas formas posibles de resolver el problema. Usando los planes generados, el diseñador completa el BTs que está construyendo actualmente. Recordemos que los BTs deben ser útiles en distintas situaciones y escenarios, por lo que este proceso que hemos descrito es en realidad un proceso iterativo donde el diseñador propone distintos escenarios al sistema y completa poco a poco el BT usando los planes generados.

Como veremos a continuación, nosotros vamos a usar ontologías para describir el vocabulario del dominio y DLPlan para resolver los problemas de planificación. Esto nos permite realizar algunas generalizaciones basadas en la jerarquía conceptual y mostrar planes abstractos aplicables en un abanico de situaciones. Por ejemplo, al calcular cómo un NPC puede matar a un enemigo usando un arma lo normal sería generar un plan distinto por cada arma que el NPC posea para representar las distintas alternativas de ataque. Sin embargo, desde el punto de vista del diseñador esta solución tiene demasiado nivel de detalle, lo único que le interesa es que el NPC puede matar al enemigo usando un arma. Podemos sacar partido de la ontología del dominio para realizar estas generalizaciones, como veremos más adelante en la sección 7.2.4.

Por último, nos gustaría aclarar que la integración de los planes como nuevas ramas del BT se realiza, a día de hoy, de manera manual. Aunque pueda parecer lo contrario, es problema no es en absoluto trivial. El planificador trabaja con una modelo del juego limitado, mientras que el diseñador juega con mucha más información (como la línea argumental del juego) para seleccionar comportamientos, modificar las precondiciones, o cambiar la prioridad de cada alternativa. La salida del planificador debe entenderse como un conjunto de propuestas que el diseñador debe validar, modificar o rechazar.

7.2.3. Conocimiento del dominio de planificación

Para poder utilizar planificación necesitamos una descripción simbólica del mundo y de las acciones que cada tipo de entidad puede efectuar. La solución más sencilla consiste en modelar el dominio desde cero. Sin embargo, esta información ya se encuentra, al menos de manera parcial, en el código C++ que los programadores deben elaborar para desarrollar el videojuego y en los ficheros de configuración que definen los distintos tipos de entidades. En esta sección explicamos nuestra propuesta, en la que partimos de un dominio parcialmente especificado y usamos componentes capaces de autodescribirse para completar la descripción simbólica del dominio.

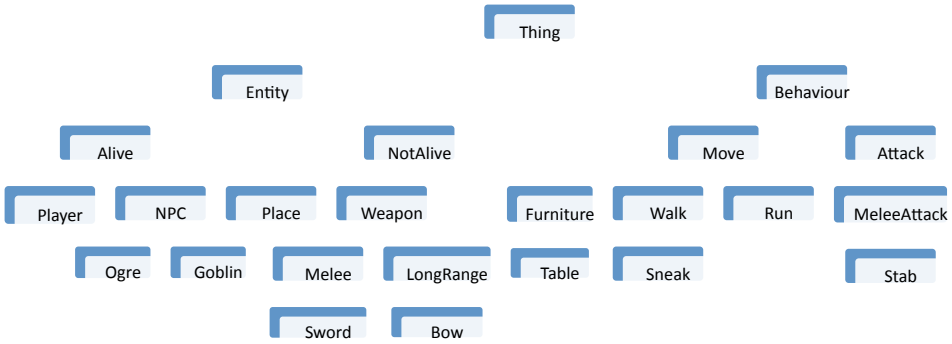


Figura 7.12: Ontología que describe los tipos de entidades del juego.

La descripción del dominio de planificación consta de dos partes: la descripción del vocabulario del dominio y los operadores de planificación. De nuevo en esta ocasión, estamos ante un dominio complejo en el que resulta natural usar ontologías para describir las distintas entidades del mundo (figura 7.12). En la parte superior de la jerarquía encontramos conceptos que representan tipos abstractos como *Entity* o *Behaviour*, que se van especializando a medida que descendemos hasta alcanzar tipos concretos como *Ogre*, *Sword* o *Stab*. Los operadores de planificación, a su vez, deben describir los comportamientos básicos que los NPCs pueden realizar, en qué condiciones del mundo se pueden llevar a cabo y qué tipo de entidades pueden ejecutarlos, y cómo se espera que transformen el estado del mundo.

Sin embargo, toda esta información ya existe, de algún modo, en el lado de la implementación. El código fuente del juego también necesita describir los distintos tipos de entidades que existen en el mundo y las acciones que pueden ejecutar. Cuando usamos la aproximación basada en componentes toda esta información está en los archivos *blueprint* y es fácil de extraer. Con respecto a los operadores de planificación, su equivalente en el lado del código son los componentes software que implementan los comportamientos básicos. Recordemos que las habilidades de cada entidad quedan determinadas por los componentes que la conforman. En nuestra propuesta este tipo de componentes deben ser capaces de auto-describirse como operadores de planificación usando precondiciones y efectos. Aunque pueda parecer que esta solución no aporta nada, tiene la ventaja de que el conocimiento se mantiene centralizado en los componentes y, por tanto, es más fácil mantener sincronizadas la implementación en C++ y su correspondiente descripción simbólica.

Comenzaremos creando una ontología del dominio sencilla que contiene

Take(?who: alive, ?what: resource)
vars: ?w, ?s
pre: canTake(?who), nextTo(?who,?what), hasWeight(?what, ?w),
hasStrength(?who, ?s), enoughStrength(?s, ?w)
del: nextTo(?who, ?what)
add: inInventary(?who, ?what)

MeleeAttack(?who: alive, ?w: weapon, ?target: alive)
vars: ?t
pre: canMeleeAttack(?who), nextTo(?who, ?target), hasWeapon(?who, ?w),
meleeWeapon(?w), hasTechnology(?w, ?t), canHandleWeaponTech(?who, ?t)
del: alive(?target)
add: dead(?target)

TakeCover(?who: alive, ?c: cover)
vars: ?r, ?s1, ?s2
pre: canTakeCover(?who), uncovered(?who), inRoom(?who, ?r), inRoom(?c, ?r),
hasSize(?who, ?s1), hasSize(?c, ?s2), lessEqSize(?s1, ?s2)
del: uncovered(?who)
add: covered(?who)

Figura 7.13: Algunos operadores asociados a los comportamientos básicos.

la terminología básica del tipo de juego que se esté creando y que debe ser suficiente para describir los operadores. Esta ontología no contiene los tipos de entidades concretos que existirán en el juego, sino la jerarquía de clases abstractas que describen tipo genéricos de entidades. Tendremos, por tanto, una jerarquía similar a la de la figura 7.12 pero sin la hojas, que las añadiremos de manera automática a partir de los ficheros *blueprint*.

Este tipo de ficheros describe cada entidad como una lista de componentes y pares atributo - valor. La descripción de nuestras entidades contiene, además, dos campos especiales *ontType* y *parentOnt* que establecen el nombre simbólico que debemos usar en la ontología para representar esta entidad y la rama o ramas en las que debemos añadirlo. También tenemos información sombre los componentes de la entidad, así que podemos iterar sobre ellos preguntándoles qué tipo de información debemos inyectar en la descripción de la entidad. Usando toda esta información podemos crear un nuevo concepto en la ontología y asociarle una fórmula lógica que describa sus propiedades. Por ejemplo, a partir de la entidad goblin que aparece en la figura 7.10 crearemos el siguiente concepto en la ontología:

$$Goblin \sqsubseteq Monster \sqcap canTake \sqcap canWalk \sqcap hasStrength.\{weak\} \ldots$$

Los operadores de planificación podemos generarlos fácilmente pidiendo a los componentes que implementan las acciones básicas que se auto-describan.

La figura 7.13 muestra algunos de los operadores de planificación que usaremos más tarde. Las precondiciones de los operadores pueden usarse para limitar el tipo de entidades que puede ejecutar cada acción. Por ejemplo, la precondición del operador *Take* tiene en cuenta la fuerza de la entidad para decidir que objetos se pueden coger, y la fuerza de cada entidad forma parte de su descripción en la ontología.

7.2.4. Ejemplo: creación de un BT

La mejor manera de comprender cómo encajan todas las piezas que hemos descrito hasta ahora es mostrando un ejemplo concreto. Durante un rato asumiremos el papel de diseñador de videojuegos y nuestro objetivo será construir un BT para controlar a un codicioso goblin que entra en una habitación y, para su regocijo, descubre un diamante grande y brillante. Supongamos también que disponemos de un interfaz amigable mediante el cual podemos definir escenarios y objetivos de manera visual, y que toda la información se traduce de manera automática a la representación simbólica que necesita el planificador.

Comenzaremos definiendo el escenario más sencillo, donde el goblin y el diamante están en la misma habitación y no hay enemigos a la vista. En particular, nuestro goblin es un guerrero y como tal está bien armado con una espada corta, un cuchillo, un arco corto y una honda. La habitación, a su vez, contiene una mesa, un par de sillas y una librería. Desde el punto de vista del planificador este estado se representa usando los siguientes hechos:

$$Goblin(goblin1), hasWeapon(goblin1, knife1), Knife(knife1),$$
$$hasWeapon(goblin1, sword1), ShortSword(sword1), \ldots,$$
$$inRoom(goblin1, room1), inRoom(table1, room1), Table(table1),$$
$$\ldots, inRoom(diamond1, room1), Diamond(diamond1)$$

El objetivo del goblin es evidente: quiere conseguir el diamante. Usando toda esta información, pedimos al planificador que calcule todas las soluciones del problema, que en este caso se reducen a una sola: caminar hasta el diamante y cogerlo.

1. `WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`

A la vista del único plan solución, construir un BT que lo represente es muy sencillo (figura 7.14). Es importante resaltar que los planes generados por el planificador son secuencias de acciones que corresponden con las hojas del árbol. La definición de los nodos internos del árbol para agrupar acciones básicas y representar distintas alternativas es responsabilidad del diseñador.

A continuación debemos completar el árbol para que sea útil en otros escenarios distintos, por ejemplo si hay un enemigo en la misma habitación

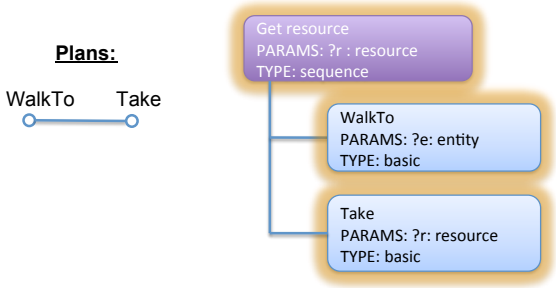


Figura 7.14: Árbol de comportamiento (primera versión).

que ha detectado al goblin. En este caso el planificador calcula varias posibles vías de acción:

1. `ChargeAt(goblin1,sword1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`
2. `ChargeAt(goblin1,knife1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`
3. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`
4. `TakeCover(goblin1,table1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`
5. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`
6. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`

Durante el cálculo de estos planes el planificador ha realizado algunas inferencias interesantes usando el conocimiento del dominio, Por ejemplo, el planificador ha usado la mesa y la librería como posibles coberturas para el goblin porque son suficientemente grandes (las sillas, por ejemplo, no lo son), y las distintas armas se han clasificado dependiente de su alcance en armas de ataque cuerpo a cuerpo o a distancia.

En realidad, los 6 planes generados son en realidad variaciones de dos estrategias usando distintos objetos: cargar contra el enemigo y coger el diamante; o buscar una cobertura, atacar al enemigo desde lejos y coger el diamante. Generalizando con cuidado los tipos de la entidades que aparecen en los 6 planes usando el conocimiento de la ontología podemos darnos cuenta de que tanto la espada corta como el cuchillo son armas de ataque cuerpo a cuerpo, que el arco y la honda son armas de ataque a distancia, y que la mesa y la librería son coberturas de tamaño grande. Tras realizar estas

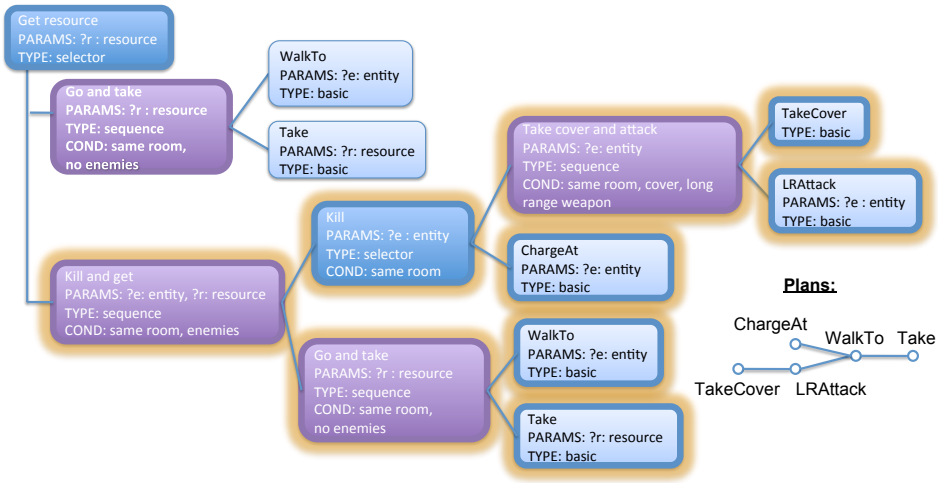


Figura 7.15: Árbol de comportamiento (segunda versión).

generalizaciones los 6 planes anteriores se transforman en dos estrategias, que son las que realmente nos muestra el sistema:

1. `ChargeAt(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamond1)`

2. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamond1), Take(goblin1,diamond1)`

Además de los dos planes generalizados, el sistema también nos muestra los estados iniciales en los que son ejecutables. El primer plan es aplicable si: *goblin1* representa una entidad que puede cargar, caminar y coger cosas; *goblin1* tiene un arma de ataque cuerpo a cuerpo llamada *sword1*; *enemy1* es una unidad enemiga que ha detectad al goblin; y *diamond1* es un objeto pequeño. El segundo plan es aplicable si *goblin1* es una entidad que puede efectuar las acciones correspondientes, tiene un arma de ataque a distancia *bow1*, y *table1* es una cobertura de tamaño mediano.

Ahora, como diseñadores, debemos modificar el BT inicial para incluir estas nuevas posibilidades. La figura 7.15 muestra el nuevo BT y las nuevas ramas aparecen iluminadas. Básicamente, el árbol anterior se ha transformado en un nodo de tipo secuencia que sólo es aplicable si no hay enemigos en la habitación. Si hay enemigos debemos eliminarles primero, y para hacerlo tenemos dos alternativas: cargar contra el enemigo o buscar una zona de cobertura y atacarle desde lejos.

Finalmente, queremos que el BT también sea aplicable si en la habitación hay un enemigo que aún no ha detectado al goblin. En este caso el planificador calcula la los siguientes planes:

1. `TakeCover(goblin1,table1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
2. `TakeCover(goblin1,table1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
3. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,bow1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
4. `TakeCover(goblin1,bookcase1), LRAttack(goblin1,sling1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
5. `SneakTo(goblin1,enemy1), MeleeAttack(goblin1,sword1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
6. `SneakTo(goblin1,enemy1), MeleeAttack(goblin1,knife1,enemy1), WalkTo(goblin1,diamon1), Take(goblin1,diamon1)`
7. `SneakTo(goblin1,diamon1), Take(goblin1,diamon1)`

Al generalizar estos planes conseguimos 3 estrategias distintas: (1) buscar cobertura, atacar desde lejos, ir hasta el diamante y cogerlo; (2) acercarse sigilosamente el enemigo, apuñalarle por la espalda, ir hasta el diamante y cogerlo; y (3) acercarse sigilosamente hasta el diamante y cogerlo sin matar al enemigo.

A la luz de estas nuevas estrategias completamos el BT y obtenemos el árbol que se muestra en la figura 7.16. En este caso la estrategia de buscar cobertura y atacar desde lejos ya estaba en el BT anterior así que sólo necesitamos añadir las otras dos posibilidades. Básicamente hemos añadido una nueva forma de matar al enemigo (*sneak and stab*) y una nueva forma de conseguir el diamante sin matar al enemigo (*sneak for resource*).

7.2.5. Conclusiones

En esta ocasión hemos utilizados técnicas de planificación para ayudar a los diseñadores de un videojuego a construir los árboles de comportamiento que controlan la IA de los personajes. En nuestro planteamiento intentamos facilitar el modelado del dominio aprovechando la arquitectura basada en componentes que se utiliza en muchos juegos modernos. Para representar el vocabulario del dominio partimos de una ontología incompleta a la que se le añaden los conceptos hoja (las entidades concretas del juego) de manera automática, procesando los ficheros *blueprints* que describen los distintos tipos de entidades. Además, los componentes que implementan los comportamientos básicos de las entidades se auto-describen como operadores de planificación usando el vocabulario de la ontología.

El proceso de construcción de BTs se realiza de forma iterativa: el diseñador plantea un escenario inicial y unos objetivos para un NPC, el planificador calcula distintos planes solución, y el diseñador integra esas soluciones como nuevas ramas del BT que está construyendo. Además, podemos generalizar

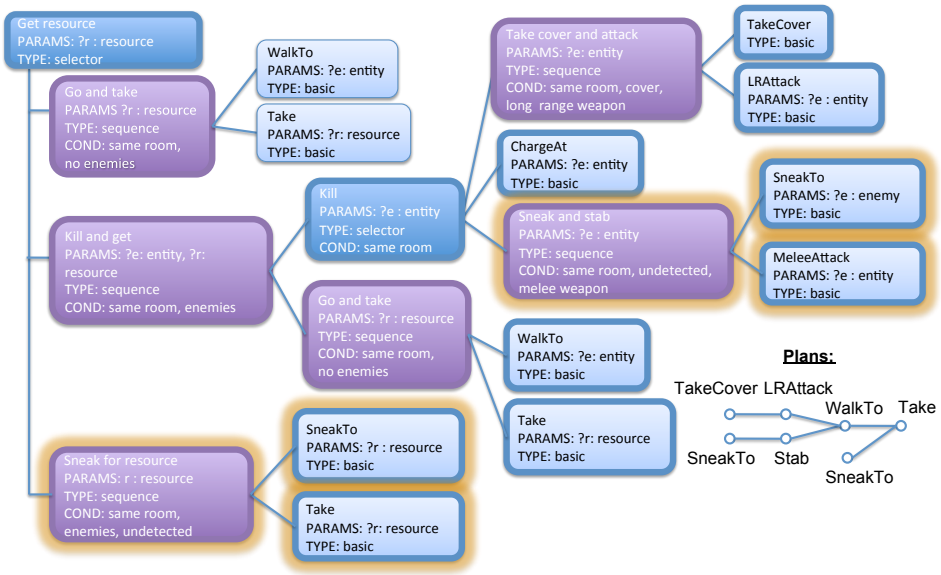


Figura 7.16: Árbol de comportamiento (versión final).

los planes solución usando la ontología del dominio para evitar abrumar al diseñador con demasiados planes con demasiado nivel de detalle.

La generalización que podemos realizar en DLPlan es limitada en dos aspectos. En primer lugar, no modificamos la estructura del plan, sólo generalizamos los tipos de las entidades que conforman el estado inicial. Esto no nos permite, por ejemplo, abstraer estrategias a partir de planes que sólo se diferencian en el orden en el que se ejecutan las acciones, incluso cuando este orden no sea importante. La segunda limitación está relacionada con la interpretación sintáctica de las acciones en DLPlan: sólo podemos subir por la jerarquía conceptual hasta cierto punto si queremos garantizar que los planes sean válidos. En nuestro caso, la generalización que podemos realizar consiste en tratar cada entidad como una instancia de los conceptos inmediatamente superiores en la ontología, que son los que se usan para describir las precondiciones y efectos de los operadores. Recordemos que las hojas de la ontología, es decir, las entidades concretas que existen en el mundo, se crean de manera automática procesando los ficheros *blueprint*, por lo que los operadores de planificación del dominio tienen que estar modelados usando conceptos más abstractos (los de la ontología incompleta inicial).

Capítulo 8

Conclusiones y trabajo futuro

Con este último capítulo concluimos la exposición del trabajo realizado. Comenzaremos haciendo un resumen de las principales ideas y propuestas que hemos desarrollado a lo largo de la memoria, a la vez que describimos las conclusiones más importantes a las que hemos llegado. A continuación, explicaremos las limitaciones más importantes del trabajo y, finalmente, propondremos algunas líneas con las que se podría continuar nuestra investigación en el futuro.

8.1. Resumen y conclusiones

Comenzamos la memoria poniendo de manifiesto el poco conocimiento que se suele incluir al modelar dominios de planificación. Estos dominios contienen dos tipos de conocimiento: el conocimiento de tipo estático, que se usa para describir la estructura del dominio y los tipos de entidades que encontramos en él, y el conocimiento de tipo dinámico, que proporciona información sobre los posibles cambios que se pueden producir en el estado del mundo. Por lo general, los modelos asociados a los dominios de planificación evitan la necesidad de proporcionar mucho conocimiento de tipo estático codificando de manea hábil los operadores de planificación. De este modo, el planificador no necesita "entender" el tipo de entidades con las que trabaja, las relaciones entre los predicados que se utilizan para representar el estado del sistema, o las restricciones del dominio que definen lo que es posible y lo que no en cada dominio. Todo ese conocimiento no es necesario porque tanto el estado del sistema como los operadores que lo modifican se describen con sumo cuidado para que no lo sea.

Esta aproximación tiene una ventaja evidente: al simplificar tanto la gestión del conocimiento los planificadores se pueden concentrar en el recorrido del espacio de búsqueda. La consecuencia es que disponemos de potentes algoritmos capaces de resolver problemas que requieren realizar un gran número de pasos en dominios sencillos, habitualmente problemas de tipo lógico

con muy poco conocimiento asociado. Entendemos por dominios sencillos aquellos que puede modelar sin mucho esfuerzo una única persona desde cero.

Sin embargo, cuando intentamos utilizar técnicas de planificación para resolver problemas del mundo real nos encontrarnos de manera natural con dominios ricos en conocimiento, dominios difíciles de modelar y con gran cantidad de información asociada. En este tipo de dominios surgen nuevos problemas relacionados con la adquisición, formalización, mantenimiento y explotación de grandes bases de conocimiento. Para poder afrontar este tipo de retos la comunidad de planificación debe trabajar en dos frentes complementarios: (1) mejorar los procesos de adquisición y gestión del conocimiento asociado a los dominios de planificación, y (2) buscar mecanismos que permitan integrar los actuales algoritmos de búsqueda y representaciones de conocimiento específico del dominio formalizado usando lenguajes expresivos.

Ante esta situación, nuestra propuesta consiste en utilizar ontologías, entendidas desde el punto de vista de la web semántica, para modelar dominios de planificación con mucho conocimiento asociado. Las ontologías fomentan la reutilización del conocimiento asociado a un dominio representando su estructura de manera intuitiva y extensible por medio de jerarquías conceptuales. Además, el uso de ontologías promueve la construcción incremental de modelos, extendiendo otros modelos relacionados creados anteriormente. Todo esto simplifica y abarata los costes asociados a los procesos de adquisición y representación del conocimiento de un dominio.

Las ontologías son uno de los pilares fundamentales para representar conocimiento en la web semántica. Debido a la cantidad y variedad del conocimiento disponible en la web, esta comunidad ha prestado especial atención a los problemas relativos a la adquisición y gestión del conocimiento, creado toda una infraestructura alrededor de estos artefactos semánticos. Como resultado de todo este esfuerzo han surgido lenguajes estándar para representar ontologías, editores visuales que facilitan su construcción, razonadores muy optimizados que permiten depurar y gestionar grandes cantidades de información de manera eficiente, repositorios de ontologías, etc.

El uso de ontologías para modelar la parte de conocimiento estática asociada a los dominios de planificación nos permite aprovechar toda esta infraestructura relacionada con la adquisición y posterior gestión de conocimiento:

- La existencia de *lenguajes estándar* y *repositorios de ontologías* facilita el modelado de dominios, y permite aprovechar el esfuerzo que otros han invertido para representar entornos similares.

- Los *editores de ontologías visuales* proporcionan entornos amigables que simplifican el acceso a estas tecnologías. Además, permiten gestionar grandes proyectos que, debido a su complejidad, serían inviables

sin estas herramientas.

- Los *razonadores* existentes nos permitirán *detectar y corregir inconsistencias* en la definición de dominios y problemas de planificación. Estos razonadores se integran en los entornos de edición de ontologías para facilitar su uso.

- La expresividad de las DLs permite definir un *vocabulario rico con el que describir información y razonar usando distintos niveles de abstracción*. Esta expresividad permite plantear problemas abstractos cuya solución puede aplicarse a multitud de problemas más específicos, y resulta especialmente interesante para aproximaciones basadas en casos.

- Los razonadores están muy optimizados y son capaces de *gestionar grandes cantidades de información de manera eficiente* y realizar interesantes inferencias a partir del conocimiento disponible.

Además, el uso de ontologías permite gestionar de manera homogénea información que proviene de distintas fuentes y compartir conocimiento entre distintos sistemas inteligentes que trabajan en el mismo dominio.

Representar el conocimiento estático de un dominio usando ontologías parece tener numerosas ventajas. Sin embargo, todo este esfuerzo resulta inútil si no disponemos de un planificador capaz de gestionar este tipo de dominios. Surgen, por tanto, dos alternativas: traducir las ontologías al lenguaje estándar de planificación, PDDL, y usar un planificador convencional; o construir un planificador que utiliza lógicas descriptivas para representar el conocimiento.

En el capítulo 3 exploramos la primera alternativa. Al traducir el conocimiento del dominio al lenguaje estándar de planificación podemos aprovechar los potentes algoritmos de búsqueda que utilizan los planificadores actuales para resolver problemas cuya solución requiere la ejecución de un gran número de acciones. La mayor parte de la veces, esta traducción se realiza de manera manual, y se necesita de un experto en las dos áreas (DLs y planificación) que detecte y extraiga el conocimiento que el planificador va a necesitar y lo represente de manera adecuada. Este proceso no sólo es difícil y laborioso, también conlleva los problemas relativos a mantener sincronizado el conocimiento representado en dos lenguajes distintos.

Los mecanismos que proporciona PDDL para representar conocimiento estático son muy básicos, por lo que durante el proceso de traducción no podemos mantener la semántica original. Aún así, hemos propuesto un algoritmo que permite realizar una traducción parcial, y que utiliza los predicados derivados para representar los conceptos intermedios y superiores de la ontología. Además, usando el lenguaje de restricciones introducido en PDDL 3

también podemos detectar inconsistencias en la definición del estado inicial de los problemas de planificación.

Las principales problemas que encontramos en la traducción están relacionados con la hipótesis de mundo abierto que se usa en OWL-DL para interpretar la información, con la posibilidad de utilizar distintos nombres para referirnos a las mismas entidades, y con la capacidad de razonamiento a partir de descripciones parciales del estado. Cuando traducimos una ontología OWL-DL a PDDL incorporamos información que no existía en la ontología original como consecuencia de "cerrar" la interpretación del mundo y asumir que las entidades tienen nombres únicos. Además, perdemos la capacidad de representar estados abstractos usando los conceptos intermedios o superiores de la ontología.

Sin embargo, incluso con todas estas limitaciones pensamos que nuestro algoritmo de traducción puede resultar útil para conseguir una primera aproximación al dominio PDDL objetivo. El resultado de la traducción debe ser revisado por un experto que, teniendo en cuenta el tipo de problemas que se pretende resolver, adaptará el conocimiento y completará el dominio con los operadores de planificación adecuados.

En el capítulo 4 comenzamos a investigar la segunda alternativa: construir un planificador que usa DLs para representar el conocimiento. Este tipo de planificación sería capaz de gestionar las ontologías del dominio manteniendo su semántica original, lo que nos permitiría resolver problemas en dominios donde la gestión del conocimiento estático juega un papel fundamental. Además, las DLs interpretan el conocimiento usando la hipótesis de mundo abierto, lo que nos permitiría resolver problemas abstractos, es decir, problemas en los que sólo se proporciona una descripción parcial del estado inicial. Finalmente, al representar el conocimiento usando DLs podríamos detectar automáticamente inconsistencias en los modelos y realizar inferencias interesantes durante el proceso de búsqueda

En este tipo de planificación el estado del sistema se representa usando una base de conocimiento DL. En realidad, lo que llamamos estado sólo es una descripción parcial del estado real del sistema, que es desconocido. Existen tantos estados reales posibles como modelos admite la base de conocimiento. En este sentido podemos decir que las DLs nos permiten representar de manera compacta el espacio de búsqueda ya que cada base de conocimiento puede representar muchas (o incluso infinitas) posibilidades. Las acciones, a su vez, se describen de manera declarativa usando precondiciones y postcondiciones. La semántica que elegimos interpreta las acciones como transformadoras de modelos, para no depender de la representación sintáctica de las bases de conocimiento.

Sin embargo, para poder construir un planificador útil basado en esta tecnología aún debemos superar algunos problemas. En primer lugar, para poder aplicar las acciones necesitamos ser capaces de representar las bases

de conocimiento actualizadas, y para ello necesitamos trabajar con la lógica $\mathcal{ALCQIO}^{@}$ que incluye nominales y el constructor @ de la lógica híbrida (o de manera equivalente Boolean ABoxes). Los razonadores actuales no permiten el uso de esta constructora. El segundo problema, mucho más complejo, consiste en tratar de adaptar los algoritmos de búsqueda que utilizan los planificadores clásicos al contexto de la planificación usando DLs. Las diferencias de expresividad de los lenguajes no hace que esto sea una tarea fácil, pero sin buenos algoritmos de búsqueda no conseguiremos transformar la planificación con DLs en una tecnología efectiva.

Trabajar con información incompleta y con teorías del dominio complejas permite describir problemas de planificación más interesantes, pero también aumenta la dificultad de los procesos de razonamiento y el potencial espacio de búsqueda. Cuando construir planes desde cero resulta costoso, podemos tratar de mejorar la eficiencia del sistema aplicando aproximaciones basadas en casos, en las que tratamos de aprovechar el esfuerzo invertido para resolver un problema cuando necesitamos resolver otro problema similar. Esta es la idea que se desarrolla en el capítulo 6, donde proponemos un sistema de planificación mixto CBR / generativo que aprovecha la riqueza expresiva de las ontologías para representar conocimiento a distintos niveles de abstracción.

La abstracción ha jugado siempre un papel esencial en la representación del conocimiento. La relación *is-a* es la base de jerarquías conceptuales donde los conceptos de la parte superior, que describen categorías abstractas, se van especializando a medida que descendemos por la jerarquía, hasta alcanzar los conceptos en la hojas que se usan para representar tipos concretos de entidades. Al representar el conocimiento usando DLs podemos valernos de la semántica formal de estos lenguajes para reducir el cálculo de la jerarquía conceptual a problemas lógicos decidibles.

La riqueza expresiva de las DLs nos permite plantear problemas con un estado inicial abstracto, donde no conocemos o no nos interesan ciertos detalles. Esta peculiaridad aumenta la aplicabilidad de los planes generados, que deben ser válidos en cualquier estado más específico que aquel que se usó para generarlos. Usando esta característica, podemos generalizar el estado inicial de un problema mediante un proceso que generalice los tipos de las entidades implicadas subiendo por la jerarquía conceptual. Esto nos permite construir casos generalizados con unas condiciones de aplicabilidad más abstractas que aquellas que se usaron para resolver el problema original.

El proceso de recuperación y adaptación de casos para resolver los nuevos problemas también se realiza usando el conocimiento del dominio y las capacidades de inferencia de un razonador. Este proceso consta de dos fases: en primer lugar recuperamos los casos cuya postcondición es más específica que el objetivo del nuevo problema, y en segundo lugar comprobamos cuales de ellos se pueden especializar de forma que sean aplicables en el nuevo estado inicial. La primera parte se resuelve usando el objetivo del problema como

consulta sobre la base de conocimiento que almacena los casos generalizados. La segunda parte se resuelve usando la precondición de los casos como consulta sobre la base de conocimiento que representa el estado inicial del nuevo problema. Al resolver estas consultas el razonador es capaz de realizar inferencias interesantes y los casos se adaptan como efecto colateral de las relaciones de abstracción entre los conceptos de la ontología del dominio.

La limitación más importante de esta aproximación es que sólo es capaz de reutilizar planes si no es necesario modificar su estructura. Por otra parte, para comprobar la idea intuitiva de que usando casos conseguimos mejorar la eficiencia del sistema necesitaríamos disponer de un planificador generativo basado en DLs con el que compararnos.

Mientras la tecnología de la planificación con DLs evoluciona, nosotros decidimos implementar un modelo simplificado en el que las acciones se interpretan de manera sintáctica. De este modo construimos el sistema DLPlan que describimos en el capítulo 5. La arquitectura de este sistema trata de separar la gestión del conocimiento estático (gestión y actualización del estado, comprobación de la consistencia del modelo, resolución de consultas, ...), que se delega en el razonador Pellet, de la gestión del proceso de búsqueda asociado al problema de planificación.

La principal limitación de este sistema reside en la interpretación sintáctica de las acciones, que deben describir de forma explícita los axiomas ABox que se deben añadir y eliminar (de manera sintáctica) de la base de conocimiento que representa el estado. Esta aproximación simplifica enormemente la actualización del estado, pero también aumenta la responsabilidad del experto que escribe los operadores, que debe tener mucho cuidado con las relaciones entre predicados y los posibles efectos colaterales de las acciones.

Estos operadores sintácticos también son dependientes de la representación sintáctica de la base de conocimiento. Por este motivo, DLPlan sólo resulta útil en dominios donde conocemos de antemano el tipo de problemas que pretendemos resolver y la manera en la que vamos a representar el estado inicial. De hecho, los planes generados sólo serán válidos en problemas que se ajustan a estas restricciones. Aún así, seguimos aprovechando la capacidad expresiva de las DLs para describir el dominio, detectar inconsistencias en la base de conocimiento, comprobar las precondiciones de los operadores, gestionar eficientemente grandes cantidades de información, etc.

Finalmente, en el capítulo 7 mostramos dos aplicaciones prácticas de nuestra tecnología en dominios con una estructura compleja donde resulta natural el uso de onotologías para modelar el conocimiento. Es importante resaltar que los problemas que vamos a resolver en estos dominios basan su dificultad, no tanto en el tamaño del espacio de búsqueda, como en la gestión del conocimiento asociado.

En primer lugar, utilizamos planificación para personalizar los ejercicios en JV^{2}M , un videojuego educativo que enseña conceptos relacionados con la

compilación de programas Java. El sistema cuenta con un corpus de ejercicios creado por expertos en el que los ejercicios están anotados con etiquetas semánticas usando el vocabulario de una ontología del dominio formalizada en OWL-DL. Cuando el módulo pedagógico del sistema necesita seleccionar el siguiente ejercicio para un estudiante, consulta la información almacenada en su perfil y selecciona los conceptos que el siguiente ejercicio debe poner el práctica y los que, debido a su dificultad, aún no deben aparecer. Usando esta información, y mediante una aproximación CBR, se elige el ejercicio del corpus más adecuado para el estudiante. Sin embargo, puede suceder que ninguno de los ejercicios disponibles sea adecuado y necesitemos adaptar alguno de ellos.

Nuestra propuesta consiste en utilizar planificación para resolver la fase de adaptación del sistema CBR, aumentando la aplicabilidad de cada caso y reduciendo, por tanto, el número de casos necesarios. La principal ventaja de esta aproximación es que sólo necesitamos describir pequeñas transformaciones atómicas como operadores y el planificador se encargará de combinarlas para realizar adaptaciones complejas. En el videojuego JV^{2}M los operadores que usamos realizan cambios sintácticos en los programas sin afectar a su semántica, lo que nos garantiza que los ejercicios adaptados sigan teniendo "sentido" para el estudiante. Finalmente, el uso de planificación basada en ontologías nos permite aprovechar la ontología del dominio que el sistema pedagógico del juego utiliza para mejorar la experiencia de aprendizaje del estudiante.

En el segundo ejemplo utilizamos técnicas de planificación para ayudar a los diseñadores de un videojuego a construir los árboles de comportamiento que controlan la IA de los personajes no jugadores. Nuestra aproximación trata de facilitar el modelado del dominio aprovechando la arquitectura basada en componentes que se utiliza en muchos juegos modernos. Para representar el vocabulario del dominio partimos de una ontología incompleta a la que se le añaden los conceptos hoja (las entidades concretas del juego) de manera automática, procesando los ficheros *blueprints* que describen los distintos tipos de entidades. Además, los componentes que implementan los comportamientos básicos de las entidades se auto-describen como operadores de planificación usando el vocabulario de la ontología.

El proceso de construcción de árboles de comportamiento se realiza de forma iterativa: el diseñador plantea un escenario inicial y unos objetivos para un personaje, el planificador calcula distintos planes solución, y el diseñador integra esas soluciones como nuevas ramas del árbol. En este caso el uso de ontologías nos permite realizar ciertas generalizaciones sobre los planes generados para evitar abrumar al diseñador con demasiados planes con demasiado nivel de detalle. Si bien los planes generalizados pueden servir de inspiración al diseñador, la integración de dichas estrategias en el árbol se realiza de forma manual. Esta decisión aumenta la responsabilidad del

diseñador pero también le da mayor control sobre el comportamiento final del personaje.

Quizá la conclusión más importante que podemos extraer de este trabajo es que aún queda mucho por hacer para conseguir técnicas de planificación que exploten de manera adecuada la enorme cantidad de conocimiento inherente a los problemas del mundo real. Para avanzar en esta dirección es necesario trabajar en dos frentes: la adquisición y gestión del conocimiento asociado a los dominios de planificación, y el desarrollo de algoritmos de búsqueda que exploten dicho conocimiento. Nosotros pensamos que el uso de ontologías pueden contribuir a resolver el primer problema. Con respecto al segundo, el lenguaje estándar de planificación ha ido evolucionando poco a poco para permitir describir problemas más interesantes. Aunque esta debe ser la línea principal de investigación, ir aumentado como a poco la expresividad del lenguaje, pensamos que también es interesante investigar el uso de representaciones más expresivas en problemas donde la dificultad reside en la gestión del conocimiento estático y no tanto en la longitud de los planes solución. Es de esperar que poco a poco ambas aproximaciones se vayan encontrando. Mientras, lo deseable es disponer de un amplio abanico de posibilidades y elegir la que mejor se adapte al problema que pretendemos resolver.

8.2. Limitaciones

El problema de la adquisición y posterior gestión del conocimiento asociado a los dominios de planificación es extremadamente complejo. Nuestra propuesta de usar ontologías para modelar la parte de conocimiento estática, aún con todas las ventajas que hemos explicado, sólo aborda una pequeña parte del problema. Recordemos que el conocimiento más importante desde el punto de vista del planificador gira en torno al conjunto de acciones disponibles y como combinarlas para construir planes. Necesitamos, por tanto, potentes herramientas que permitan gestionar todo el conocimiento (estático y dinámico) asociado a un dominio de manera conjunta, y que ayuden al usuario durante todo el proceso de modelado. Siguiendo la filosofía de las ontologías, que permiten modelar dominios complejos de forma modular extendiendo otras ontologías existentes, sería deseable poder hacer lo mismo con los dominios de planificación, y poder reutilizar no sólo la parte de conocimiento estática sino también los operadores y el conocimiento asociado a la búsqueda.

Además, una visión global de los procesos de Ingeniería del Conocimiento aplicados a la planificación también debe tener en cuenta otros aspectos como los distintos tipos de personas implicadas en la construcción de un sistema de planificación (expertos en planificación, expertos del dominio, programadores, usuarios finales, . . .) y los distintos flujos de comunicación y colaboración

entre ellos. Finalmente, pero no menos importante, es necesario aprender a caracterizar bien los dominios y las distintas técnicas de planificación para elegir la más adecuada en cada caso.

También queda por resolver el problema de la planificación usando DLs para representar el conocimiento. Aunque ya existen modelos teóricos que permiten representar y recorrer el espacio de estados usando estos formalismos, aún queda por desarrollar la tecnología necesaria para hacerlo. También necesitamos encontrar maneras inteligentes de representar y recorrer el espacio de búsqueda si queremos ser capaces de afrontar la explosión de posibilidades inherente a los problemas de planificación.

Al usar lenguajes más expresivos para modelar el conocimiento aumenta la dificultad de los problemas de planificación. Sin embargo, el uso de lenguajes expresivos también facilita la representación de modelos del dominio con mucho conocimiento asociado que se podría explotar para resolver los problemas. Esto es especialmente interesante en problemas donde la solución es evidente sólo si se "comprenden" los conceptos implicados en la definición del dominio.

Con respecto al uso de casos para mejorar la eficiencia de un planificador generativo basado en DLs, queda por demostrar que nuestra intuición se cumple en la práctica. Además, nuestra propuesta se basa en utilizar la ontología del dominio y las capacidades de inferencia de un razonador para generalizar los problemas, comprobar si los objetivos de un nuevo problema son compatibles con las postcondiciones de alguno de los casos, y adaptar los planes de dichos casos al contexto inicial del nuevo problema. Esta aproximación está muy limitada porque sólo permite adaptar planes si no es necesario modificar su estructura. Quedan por investigar otras aproximaciones CBR más flexibles que permitan hacer adaptaciones más interesantes, seguramente basadas en analogía por derivación (guardar las decisiones que llevaron a la solución del caso y tomar las mismas decisiones en el contexto del nuevo problema).

Finalmente, nosotros hemos combinado con éxito planificación y ontologías en dos dominios concretos: la adaptación de ejercicios en el videojuego educativo JV^{2}M , y la ayuda durante el proceso de definición del comportamiento de los personajes no jugadores de un videojuego. Sería interesante comprobar cómo se aplican todas estas ideas en otros dominios distintos.

8.3. Trabajo futuro

Cada una de las limitaciones que hemos descrito en el apartado anterior abre una posible línea de investigación con la que mejorar este trabajo. Sin embargo, en este apartado sólo hablare de las líneas de trabajo futuro que veo más probable acometer teniendo en cuenta tanto mis preferencias personales como el trabajo que se lleva a cabo actualmente en mi grupo de investigación.

En primer lugar, sería interesante profundizar en el papel que juegan las ontologías en los sistemas de planificación basada en casos. La ontologías nos brindan la posibilidad de describir conocimiento a distintos niveles de abstracción, permiten calcular similitudes que explotan la estructura del dominio, y abren las puertas a interesantes técnicas de adaptación más allá de las que hemos descrito en este trabajo. Además, el uso de ontologías como base de casos también puede resultar útil desde el punto de vista de la compartición de conocimiento, por ejemplo cuando se necesita trasladar conocimiento entre dominios similares.

Las ontologías también son una prometedora fuente de conocimiento que podría utilizarse para definir heurísticas que guíen el proceso de búsqueda. Usando medidas de similitud que exploten la estructura del dominio podemos calcular con mayor exactitud la similitud entre un estado intermedio de la búsqueda y un estado objetivo, priorizando aquellos caminos que nos acercan a la solución. Además, de este modo podríamos proporcionar planes, que si bien no resuelven el problema planteado, al menos nos llevan hasta un estado similar al que buscábamos. Esto puede resultar interesante en ciertos dominios con objetivos "blandos" donde basta acercarse al objetivo planteado, o en dominios con objetivos demasiado difíciles en los que casi nunca se encuentran soluciones reales, o en sistemas interactivos donde el usuario puede modificar los objetivos del problema hasta alcanzar un plan que le gusta.

Las ontologías también pueden ayudar a mejorar la comunicación con el usuario en sistemas interactivos, proporcionando mejores explicaciones sobre las decisiones tomadas o sobre las diferencias entre las soluciones disponibles. Esto puede ser especialmente interesante en sistemas de planificación mixta (primeros principios y basada en casos), que pueden resultar útiles en dominios reales demasiado complejos para modelar en su totalidad. La componente generativa se encargaría de resolver los problemas relacionados con las zonas del dominio que se comprenden mejor, mientras los casos contienen conocimiento del dominio que no sabemos cómo modelar. Este tipo de sistemas suele ser interactivo y la colaboración con el usuario resulta fundamental a la hora de resolver problemas.

Otra idea que puede resultar interesante consiste en tratar de interpretar el conocimiento que se representa en las ontologías bajo la hipótesis de mundo cerrado usando un proceso interactivo donde el usuario valida el conocimiento que se añade al modelo. El objetivo sería poder traducir los estados abstractos como conjuntos finitos de posibles estados concretos y de ese modo poder utilizar alguno de los planificadores de tipo *conformant* para resolver problemas. De este modo podríamos combinar cierto grado de indeterminismo en el estado inicial con un buen rendimiento del planificador a la hora de resolver problemas.

Finalmente, nos resulta interesante la combinación de ontologías y plani-

ficación en dominios relacionados con juegos [132, 134]. En los videojuegos se trabaja con mundos complejos, muy estructurados, en los que existen multitud de entidades distintas, y las ontologías son una manera intuitiva de representar todo este conocimiento. Nos gustaría seguir investigando la aplicación de técnicas interactivas en las que se usa técnicas de planificación para crear mejores herramientas de autoría. En estos casos el objetivo principal no es tanto el rendimiento del planificador como la capacidad de describir las soluciones de forma intuitiva para el usuario. Estas ideas se pueden utilizar para mejorar la creación de contenidos en videojuegos, entornos virtuales y juegos educativos.

Bibliografía

[1] Jena - A Semantic Web Framework for Java.`http://jena.sourceforge.net/index.html`. (Visitado en Abril de 2010).

[2] Aamodt, A. y Plaza, E. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Commun.*, vol. 7(1), páginas 39–59, 1994.

[3] Althoff, K.-D., Bergmann, R., Minor, M. y Hanft, A., editores. *Advances in Case-Based Reasoning, 9th European Conference, ECCBR 2008, Trier, Germany, September 1-4, 2008. Proceedings*, vol. 5239 de *Lecture Notes in Computer Science*. Springer, 2008. ISBN 978-3-540-85501-9.

[4] Arias, J. D., Sebastiá, L. y Borrajo, D. Using Ontologies for Planning Tourist Visits. En *Working notes of the ICAPS'05 Workshop on Role of Ontologies in Planning and Scheduling* (editado por J. F. Olivares y E. Onaindía), páginas 52–59. AAAI Press, Monterey, CA (EEUU), 2005.

[5] Au, T.-C., Mu H., noz-Avila y Nau, D. S. On the Complexity of Plan Adaptation by Derivational Analogy in a Universal Classical Planning Framework. En *ECCBR '02: Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, páginas 13–27. Springer-Verlag, London, UK, 2002. ISBN 3-540-44109-3.

[6] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. y Patel-Schneider, P. F., editores. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, New York, NY, USA, 2003. ISBN 0-521-78176-0.

[7] Baader, F., Lutz, C. y Suntisrivaraporn, B. CEL—a polynomial-time reasoner for life science ontologies. En *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)* (editado por U. Furbach y N. Shankar), vol. 4130 de *Lecture Notes in Artificial Intelligence*, páginas 287–291. Springer-Verlag, 2006.

[8] BAADER, F., MILICIC, M., LUTZ, C., SATTLER, U. y WOLTER, F. Integrating Description Logics and Action Formalisms: First Results. En *Description Logics* (editado por I. Horrocks, U. Sattler y F. Wolter), vol. 147 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[9] BACCHUS, F. Subset of PDDL for the AIPS2000 Planning Competition. Informe técnico, The AIPS-00 Planning Competition Committee, 2000.

[10] BACCHUS, F. y KABANZA, F. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artif. Intell.*, vol. 116(1-2), páginas 123–191, 2000. ISSN 0004-3702.

[11] BARTÁK, R. y MCCLUSKEY, L. The First Competition on Knowledge Engineering for Planning and Scheduling. *AI Magazine*, vol. 27(1), páginas 97–98, 2006.

[12] BARTÁK, R., FRATINI, S. y MCCLUSKEY, L., editores. *Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling*. Thessaloniki, Greece, 2009.

[13] BECKETT, D. RDF/XML Syntax Specification (Revised). W3C recommendation, W3C, 2004. Http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[14] BERGMANN, R. y WILKE, W. Building and Refining Abstract Planning Cases by Change of Representation Language. *J. Artif. Int. Res.*, vol. 3(1), páginas 53–118, 1995. ISSN 1076-9757.

[15] BERNERS-LEE, T., FIELDING, R. y MASINTER, L. Uniform Resource Identifier (URI): Generic Syntax. Informe Técnico RFC 3986, The Internet Society, 2005. Http://www.ietf.org/rfc/rfc3986.txt.

[16] BERNERS-LEE, T., HENDLER, J. y LASSILA, O. The Semantic Web. *Scientific American*, vol. 284(5), páginas 34–43, May 2001.

[17] BIUNDO, S., AYLETT, R., BEETZ, M., BORRAJO, D., CESTA, A., GRANT, T., MCCLUSKEY, T., MILANI, A. y VERFAILLE, G. Technological Roadmap on AI Planning and Scheduling. Informe Técnico IST-2000-29656, PLANET, the European Network of Excellence in AI Planning, 2003.

[18] BIUNDO, S. y SCHATTENBERG, B. From abstract crisis to concrete relief - A preliminary report on combining state abstraction and HTN planning. En *In Proceedings of the European Conference on Planning*, páginas 157–168. Springer Verlag, 2001.

[19] BODDY, M. S., FOX, M. y THIÉBAUX, S., editores. *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*. AAAI, 2007. ISBN 978-1-57735-344-7.

[20] BONET, B. y GEFFNER, H. Planning as heuristic search. *Artif. Intell.*, vol. 129(1-2), páginas 5–33, 2001.

[21] BONG, Y. *Description Logic ABox Updates Revisited*. Master thesis, TU Dresden, Germany, 2007.

[22] BOUILLET, E., FEBLOWITZ, M., LIU, Z., RANGANATHAN, A. y RIABOV, A. A Knowledge Engineering and Planning Framework based on OWL Ontologies. En *ICAPS'07 Knowledge Engineering Competition (ICKEPS)*. 2007.

[23] BRACHMAN, R. y LEVESQUE, H. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 1558609326.

[24] VAN DEN BRAND, M. G. J. y KLINT, P. ATerms for manipulation and exchange of structured data: It's all about sharing. *Inf. Softw. Technol.*, vol. 49(1), páginas 55–64, 2007. ISSN 0950-5849.

[25] BRICKLEY, D. y GUHA, R. V. RDF Vocabulary Description Language 1.0: RDF Schema. W3C recommendation, W3C, 2004. Http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[26] BRYCE, D. y BUFFET, O. 6th International Planning Competition: Uncertainty Part. Informe técnico, The IPC-08 Planning Competition Committee, 2008.

[27] BUCHANAN, W. *Game Programming Gems 5*, capítulo A Generic Component Library. Charles River Media, 2005. ISBN 1-584-50352-1.

[28] CARBONELL, J. *Machine Learning: An Artificial Intelligence Approach*, capítulo Learning by analogy: Formulating and generalizing plans from past experience, páginas 371–392. Tioga Publishing, Palo Alto, 1983.

[29] CARBONELL, J. *Machine Learning: An Artificial Intelligence Approach: Volumen II*, capítulo Derivational analogy: A theory of reconstructive problem solving and expertise acquisition, páginas 371–392. Morgan Kaufmann, Los Altos, 1986.

[30] CURRIE, K. y TATE, A. O-Plan: The Open Planning Architecture. *Artif. Intell.*, vol. 52(1), páginas 49–86, 1991.

[31] DAVIS, R., SHROBE, H. E. y SZOLOVITS, P. What Is a Knowledge Representation? *AI Magazine*, vol. 14(1), páginas 17–33, 1993.

[32] DOHERTY, P., MYLOPOULOS, J. y WELTY, C. A., editores. *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*. AAAI Press, 2006. ISBN 978-1-57735-271-6.

[33] DUERST, M. y SUIGNARD, M. Internationalized Resource Identifiers (IRIs). Informe Técnico RFC 3987, The Internet Society, 2005. Http://www.ietf.org/rfc/rfc3987.txt.

[34] DÍAZ-AGUDO, B., GONZÁLEZ-CALERO, P. A., RECIO-GARCÍA, J. A. y SÁNCHEZ-RUIZ, A. A. Building CBR systems with jCOLIBRI. *Special Issue on Experimental Software and Toolkits of the Journal Science of Computer Programming*, vol. 69(1-3), páginas 68–75, 2007. ISSN 0167-6423.

[35] DÍAZ-AGUDO, B., PLAZA, E., RECIO-GARCÍA, J. A. y ARCOS, J. L. Noticeably New: Case Reuse in Originality-Driven Tasks. En [3], páginas 165–179.

[36] EDELKAMP, S. y HOFFMANN, J. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Informe técnico, 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2004.

[37] EROL, K., HENDLER, J. A. y NAU, D. S. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. En *AIPS*, páginas 249–254. 1994.

[38] EROL, K., HENDLER, J. A. y NAU, D. S. Complexity Results for HTN Planning. *Ann. Math. Artif. Intell.*, vol. 18(1), páginas 69–93, 1996.

[39] FALLSIDE, D. C. y WALMSLEY, P. XML Schema Part 0: Primer Second Edition. W3C recommendation, W3C, 2004. Http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[40] FERNÁNDEZ-OLIVARES, J. y ONAINDÍA, E., editores. *Workshop on the Role of Ontologies in Planning and Scheduling*. Monterey, California, USA, 2005.

[41] FINGER, J. J. *Exploiting Constraints in Design Synthesis*. Tesis Doctoral, Stanford University, Stanford, CA, USA, 1987.

[42] FOX, M. y LONG, D. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, vol. 20, 2003.

[43] GARCÉS, S. *AI Game Programming Wisdom 3*, capítulo Flexible Object-Composition Architecture. Charles River Media, 2006. ISBN 1-584-50457-9.

[44] GENNARI, J. H., MUSEN, M. A., FERGERSON, R. W., GROSSO, W. E., CRUBÉZY, M., ERIKSSON, H., NOY, N. F. y TU, S. W. The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. *Int. J. Hum.-Comput. Stud.*, vol. 58(1), páginas 89–123, 2003. ISSN 1071-5819.

[45] GEREVINI, A. y LONG, D. Preferences and Soft Constraints in PDDL3. En *ICAPS-2006 Workshop on Preferences and Soft Constraints in Planning*, páginas 46–54. 2006.

[46] GHALLAB, M., ISI, C. K., PENBERTHY, S., SMITH, D. E., SUN, Y. y WELD, D. PDDL - The Planning Domain Definition Language. Informe técnico, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[47] GIL, Y. Description Logics and Planning. *AI Magazine*, vol. 26(2), páginas 73–84, 2005.

[48] GRUBER, T. R. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, vol. 5(2), páginas 199 – 220, 1993. ISSN 1042-8143.

[49] GÓMEZ-MARTÍN, M. A., GÓMEZ-MARTÍN, P. P. y GONZÁLEZ-CALERO, P. A. Aprendizaje activo en simulaciones interactivas. *Revista Iberoamericana de Inteligencia Artificial*, vol. 11(33), páginas 25–36, 2007. ISSN 1137-3601.

[50] GÓMEZ-MARTÍN, M. A., GÓMEZ-MARTÍN, P. P., PALMIER CAMPOS, P. y GONZÁLEZ-CALERO, P. A. Not Yet Another Visualization Tool: Learning Compilers for Fun. En *8th International Symposium on Computers in Education (SIIE'06)* (editado por L. Panizo-Alonso, L. Sánchez-González, B. Fernández-Manjón y M. Llamas-Nistal), páginas 264–271. Universidad de León, León, Spain, 2006. ISBN 84-9773-301-0.

[51] GÓMEZ-MARTÍN, P. P., GÓMEZ-MARTÍN, M. A., GONZÁLEZ-CALERO, P. A. y PALMIER-CAMPOS, P. Using Metaphors in Game-Based Education. En *Technologies for E-Learning and Digital Entertainment. Second International Conference of E-Learning and Games (Edutainment'07)* (editado por K. chuen Hui, Z. Pan, R. C. kit Chung, C. C. Wang, X. Jin, S. Göbel y E. C.-L. Li), vol. 4469 de *Lecture Notes in Computer Science*, páginas 477–488. Springer Verlag, 2007. ISBN 3-540-73010-9.

[52] HAARSLEV, V. y MÖLLER, R. Description of the RACER System and its Applications. En *Proceedings International Workshop on Description Logics (DL-2001), Stanford, USA, 1.-3. August*, páginas 131–141. 2001.

[53] HAARSLEV, V. y MÖLLER, R. The DIG Description Logic Interface. En *Proceedings of the International Workshop on Description Logics (DL-2003), Rome, Italy, September 5-7*. 2003.

[54] HALASCHEK-WIENER, C. y HENDLER, J. A. Toward Expressive Syndication on the Web. En *WWW* (editado por C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider y P. J. Shenoy), páginas 727–736. ACM, 2007. ISBN 978-1-59593-654-7.

[55] HALASCHEK-WIENER, C., PARSIA, B. y SIRIN, E. Description Logic Reasoning with Syntactic Updates. En *OTM Conferences (1)* (editado por R. Meersman y Z. Tari), vol. 4275 de *Lecture Notes in Computer Science*, páginas 722–737. Springer, 2006. ISBN 3-540-48287-3.

[56] HALASCHEK-WIENER, C., PARSIA, B., SIRIN, E. y KALYANPUR, A. Description Logic Reasoning for Dynamic ABoxes. En [108].

[57] HAMMOND, K. J. *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press Professional, Inc., San Diego, CA, USA, 1989. ISBN 0-12-322060-2.

[58] HANKS, S. y WELD, D. S. A Domain-Independent Algorithm for Plan Adaptation. *J. Artif. Int. Res.*, vol. 2(1), páginas 319–360, 1994. ISSN 1076-9757.

[59] VAN HARMELEN, F. y MCGUINNESS, D. L. OWL Web Ontology Language Overview. W3C recommendation, W3C, 2004. Http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[60] VAN HEIJST, G., SCHREIBER, A. T. y WIELINGA, B. J. Using Explicit Ontologies in KBS Development. *Int. J. Hum.-Comput. Stud.*, vol. 46(2), páginas 183–292, 1997.

[61] HELMERT, M. The Fast Downward Planning Systems. *J. Artif. Int. Res.*, vol. 26(1), páginas 191–246, 2006. ISSN 1076-9757.

[62] HELMERT, M. Changes in PDDL 3.1.`http://ipc.informatik.uni-freiburg.de/PddlExtension`. (Visitado en Abril de 2010).

[63] HOFFMANN, J. FF: The Fast-Forward Planning System. *AI Magazine*, vol. 22(3), páginas 57–62, 2001.

[64] HORRIDGE, M. A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools. Edition 1.2. Informe técnico, The University Of Manchester, 2009.

[65] HORRIDGE, M. TONES Ontology Repository.`http://owl.cs.manchester.ac.uk/repository/`. (Visitado en Abril de 2010).

[66] HORRIDGE, M. y BECHHOFER, S. The owl api: A java api for working with owl 2 ontologies. En *OWLED* (editado por R. Hoekstra y P. F. Patel-Schneider), vol. 529 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[67] HUSTADT, U., MOTIK, B. y SATTLER, U. Reducing $\mathcal{SHIQ}^-$ Description Logic to Disjunctive Datalog Programs. En *Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR2004)* (editado por D. Dubois, C. Welty y M.-A. Williams), páginas 152–162. AAAI Press, 2004.

[68] IHRIG, L. H. y KAMBHAMPATI, S. Derivation Replay for Partial-Order Planning. En *AAAI*, páginas 992–997. 1994.

[69] ISLA, D. Handling Complexity in the Halo 2 AI. En *Game Developers Conference*. 2005.

[70] ISLA, D. Halo 3 - Building a Better Battle. En *Game Developers Conference*. 2008.

[71] KALYANPUR, A. *Debugging and repair of owl ontologies*. Tesis Doctoral, University of Maryland, College Park, MD, USA, 2006. Adviser-Hendler, James.

[72] KAMBHAMPATI, S. y HENDLER, J. A. A Validation-Structure-Based Theory of Plan Modification and Reuse. *Artif. Intell.*, vol. 55(2), páginas 193–258, 1992.

[73] KASHYAP, V., BUSSLER, C. y MORAN, M. *The Semantic Web. Semantics for Data and Services on the Web*, capítulo 6. Springer, 2008.

[74] KATSUNO, H. y MENDELZON, A. O. On the Difference between Updating a Knowledge Base and Revising It. En *KR*, páginas 387–394. 1991.

[75] KETTLER, B. P., HENDLER, J. A., ANDERSEN, W. A. y EVETT, M. P. Massively Parallel Support for Case-Based Planning. *IEEE Expert: Intelligent Systems and Their Applications*, vol. 9(1), páginas 8–14, 1994. ISSN 0885-9000.

[76] KNUBLAUCH, H., MUSEN, M. A. y RECTOR, A. L. Editing Description Logic Ontologies with the Protege OWL Plugin. En *Proc. of International Conf. on Description Logics*. 2004.

[77] LAKOS, J. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.

[78] LEAKE, D. B., KINLEY, A. y WILSON, D. C. A Case Study of Case-Based CBR. En *ICCBR*, páginas 371–382. 1997.

[79] LINDBERG, D., HUMPHREYS, B. y MCCRAY, A. The Unified Medical Language System. *Methods of Information in Medicine*, vol. 32(4), páginas 281–291, 1993.

[80] LIU, H., LUTZ, C., MILICIC, M. y WOLTER, F. DL Actions with GCIs: a Pragmatic Approach. En [108].

[81] LIU, H., LUTZ, C., MILICIC, M. y WOLTER, F. Reasoning About Actions Using Description Logics with General TBoxes. En *JELIA* (editado por M. Fisher, W. van der Hoek, B. Konev y A. Lisitsa), vol. 4160 de *Lecture Notes in Computer Science*, páginas 266–279. Springer, 2006. ISBN 3-540-39625-X.

[82] LIU, H., LUTZ, C., MILICIC, M. y WOLTER, F. Updating Description Logic ABoxes. En [32], páginas 46–56.

[83] MAXIMINI, K., MAXIMINI, R. y BERGMANN, R. An Investigation of Generalized Cases. En *ICCBR* (editado por K. D. Ashley y D. G. Bridge), vol. 2689 de *Lecture Notes in Computer Science*, páginas 261–275. Springer, 2003. ISBN 3-540-40433-3.

[84] MCCARTHY, J. Epistemological Problems of Artificial Intelligence. En *IJCAI'77: Proceedings of the 5th international joint conference on Artificial intelligence*, páginas 1038–1044. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1977.

[85] MCCARTHY, J. y HAYES, P. J. Some Philosophical Problems from the Standpoint of Artificial Intelligence. En *Machine Intelligence 4* (editado por B. Meltzer y D. Michie), páginas 463–502. Edinburgh University Press, 1969. Reprinted in McC90.

[86] MCCLUSKEY, L. y CRESSWELL, S. Importing Ontological Information into Planning Domain Models. En *Working notes of the ICAPS'05 Workshop on Role of Ontologies in Planning and Scheduling* (editado por J. F. Olivares y E. Onaindía), páginas 5–12. AAAI Press, Monterey, CA (USA), 2005.

[87] MCCLUSKEY, T. L. PDDL: A Language with a Purpose? En *ICAPS-2003 Workshop on PDDL, International Conference on Automated Planning and Scheduling*. Trento, Italy., 2003.

[88] MCDERMOTT, D. Using regression-match graphs to control search in planning. *Artif. Intell.*, vol. 109(1-2), páginas 111–159, 1999. ISSN 0004-3702.

[89] MCDERMOTT, D. V. y DOU, D. Representing Disjunction and Quantifiers in RDF. En *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, páginas 250–263. Springer-Verlag, London, UK, 2002. ISBN 3-540-43760-6.

[90] MCNEILL, F., BUNDY, A. y WALTON, C. Planning from Rich Ontologies through Translation Between Representations. En *Working notes of the ICAPS'05 Workshop on Role of Ontologies in Planning and Scheduling* (editado por J. F. Olivares y E. Onaindía), páginas 13–21. AAAI Press, Monterey, CA (EEUU), 2005.

[91] MILICIC, M. Complexity of Planning in Action Formalisms Based on Description Logics. En *LPAR* (editado por N. Dershowitz y A. Voronkov), vol. 4790 de *Lecture Notes in Computer Science*, páginas 408–422. Springer, 2007. ISBN 978-3-540-75558-6.

[92] MILICIC, M. Planning in Action Formalisms based on DLs: First Results. En *Description Logics* (editado por D. Calvanese, E. Franconi, V. Haarslev, D. Lembo, B. Motik, A.-Y. Turhan y S. Tessaris), vol. 250 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[93] MORISSET, B. y GHALLAB, M. Learning How to Combine Sensory-Motor Modalities for a Robust Behavior. En *Revised Papers from the International Seminar on Advances in Plan-Based Control of Robotic Agents,*, páginas 157–178. Springer-Verlag, London, UK, 2002. ISBN 3-540-00168-9.

[94] MOTIK, B., SHEARER, R. y HORROCKS, I. Optimized Reasoning in Description Logics using Hypertableaux. En *Proc. of the 21st Conference on Automated Deduction (CADE-21)* (editado por F. Pfenning), vol. 4603 de *LNAI*, páginas 67–83. Springer, Bremen, Germany, 2007.

[95] MUÑOZ-AVILA, H., AHA, D. W., BRESLOW, L. y NAU, D. S. HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations. En *AAAI/IAAI*, páginas 870–875. 1999.

[96] MUÑOZ-AVILA, H., AHA, D. W., NAU, D. S., WEBER, R., BRESLOW, L. y YAMAN, F. SiN: Integrating Case-based Reasoning with Task

Decomposition. En *IJCAI* (editado por B. Nebel), páginas 999–1004. Morgan Kaufmann, 2001. ISBN 1-55860-777-3.

[97] MUÑOZ-AVILA, H. y HOANG, H. *AI Game Programing Wisdom 3*, capítulo Coordinating Teams of Bots with Hierarchical Task Network Planning. Charles River Media, 2006.

[98] MUÑOZ-AVILA, H. y WEBERSKIRCH, F. Planning for Manufacturing Workpieces by Storing, Indexing and Replaying Planning Decisions. En *AIPS*, páginas 150–157. 1996.

[99] MUSEN, M. The Protégé Ontology Editor and Knowledge Acquisition System.`http://protege.stanford.edu/`. (Visitado en Abril de 2010).

[100] NAU, D., GHALLAB, M. y TRAVERSO, P. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 1558608567.

[101] NAU, D. S., AU, T.-C., ILGHAMI, O., KUTER, U., MURDOCK, J. W., WU, D. y YAMAN, F. SHOP2: An HTN Planning System. *J. Artif. Intell. Res. (JAIR)*, vol. 20, páginas 379–404, 2003.

[102] NAU, D. S., CAO, Y., LOTEM, A. y MUÑOZ-AVILA, H. SHOP: Simple Hierarchical Ordered Planner. En *IJCAI* (editado por T. Dean), páginas 968–975. Morgan Kaufmann, 1999. ISBN 1-55860-613-0.

[103] NEBEL, B. y KOEHLER, J. Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. *Artif. Intell.*, vol. 76(1-2), páginas 427–454, 1995. ISSN 0004-3702.

[104] NEWTON, M. HIBERNATE. Relational Persistence for Java and .NET.`http://www.hibernate.org/`. (Visitado en Abril de 2010).

[105] OKHTAY, O. y NAU, D. A General Approach to Synthesize Problem-Specific Planners. Informe Técnico CS-TR-4597, UMIACS-TR-2004-40, University of Maryland, 2003.

[106] PAOLI, J., COWAN, J., BRAY, T., YERGEAU, F., MALER, E. y SPERBERG-MCQUEEN, C. M. Extensible Markup Language (XML) 1.1 (Second Edition). W3C recommendation, W3C, 2006. Http://www.w3.org/TR/2006/REC-xml11-20060816.

[107] PARSIA, B., HALASCHEK-WIENER, C. y SIRIN, E. Towards Incremental Reasoning Through Updates in OWL-DL. En *Reasoning on the Web Workshop-WWW2006, Edinburgh, Scotland, UK*. 2006.

[108] PARSIA, B., SATTLER, U. y TOMAN, D., editores. *Proceedings of the 2006 International Workshop on Description Logics (DL2006), Windermere, Lake District, UK, May 30 - June 1, 2006*, vol. 189 de *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[109] RABIN, S., editor. *AI Game Programming Wisdom 3*. Charles River Media, 2006.

[110] RABIN, S., editor. *AI Game Programming Wisdom 4*. Charles River Media, 2008.

[111] RECIO-GARCÍA, J. A. *jCOLIBRI: A multi-level platform for building and generating CBR systems*. Tesis Doctoral, Universidad Complutense de Madrid, 2008.

[112] RECIO-GARCÍA, J. A., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Textual CBR in jCOLIBRI: From Retrieval to Reuse. En *Proceedings of the ICCBR 2007 Workshop on Textual Case-Based Reasoning: Beyond Retrieval.* (editado por D. C. Wilson y D. Khemani), páginas 217–226. 2007. ISBN 978-1-85923-22-4.

[113] RECIO-GARCÍA, J. A., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Semantic Templates for Case-Based Reasoning Systems. *The Knowledge Engineering Review*, vol. 24(Special Issue 03), páginas 245–264, 2009. ISSN 0269-8889.

[114] RECIO-GARCÍA, J. A., DÍAZ-AGUDO, B., GONZÁLEZ-CALERO, P. A. y SÁNCHEZ-RUIZ, A. A. Ontology based CBR with jCOLIBRI. En *Applications and Innovations in Intelligent Systems XIV. Proceedings of AI-2006, the Twenty-sixth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (editado por R. Ellis, T. Allen y A. Tuson), páginas 149–162. Springer, Cambridge, United Kingdom, 2006. ISBN 1-84628-665-4.

[115] RECIO-GARCÍA, J. A., DÍAZ-AGUDO, B., GÓMEZ-MARTÍN, M. A. y WIRATUNGA, N. Extending jCOLIBRI for Textual CBR. En *ICCBR* (editado por H. Muñoz-Avila y F. Ricci), vol. 3620 de *Lecture Notes in Computer Science*, páginas 421–435. Springer, 2005. ISBN 3-540-28174-6.

[116] RECIO-GARCÍA, J. A., DÍAZ-AGUDO, B., SÁNCHEZ-RUIZ, A. A. y GONZÁLEZ-CALERO, P. A. Lessons Learnt in the Development of a CBR Framework. En *Proccedings of the 11th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 60–71. CMS Press,University of Greenwich, 2006. ISBN 987-1-904521-39-6.

[117] RECIO-GARCÍA, J. A., SÁNCHEZ-RUIZ, A. A., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. jCOLIBRI 1.0 in a nutshell. A software tool for designing CBR systems. En *Proccedings of the 10th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 20–28. CMS Press,University of Greenwich, 2005. ISBN 1-904521-30-4.

[118] RECTOR, A. L., NOWLAN, W. A., KAY, S., GOBLE, C. A. y HOWKINS, T. J. A Framework for Modelling the Electronic Medical Record. *Methods of Information in Medicine*, vol. 32, páginas 109–119, 1993.

[119] RENE, B. *Game Programming Gems 5*, capítulo Component Based Object Management. Charles River Media, 2005. ISBN 1-584-50352-1.

[120] RINTANEN, J., NEBEL, B., BECK, J. C. y HANSEN, E. A., editores. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*. AAAI, 2008. ISBN 978-1-57735-386-7.

[121] SACERDOTI, E. D. The Nonlinear Nature of Plans. En *IJCAI*, páginas 206–214. 1975.

[122] SEABORNE, A. y PRUD'HOMMEAUX, E. SPARQL Query Language for RDF. W3C recommendation, W3C, 2008. Http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[123] SIRIN, E. *Combining Description Logic Reasoning with AI Planning for Composition of Web Services*. Tesis Doctoral, Computer Science Department of University of Maryland (USA), 2006.

[124] SIRIN, E., GRAU, B. C. y PARSIA, B. From Wine to Water: Optimizing Description Logic Reasoning for Nominals. En [32], páginas 90–99.

[125] SIRIN, E. y PARSIA, B. Optimizations for Answering Conjunctive ABox Queries. En [108].

[126] SIRIN, E., PARSIA, B., GRAU, B. C., KALYANPUR, A. y KATZ, Y. Pellet: A Practical OWL-DL Reasoner. *Web Semant.*, vol. 5(2), páginas 51–53, 2007. ISSN 1570-8268.

[127] SMITH, S. J. J., NAU, D. S. y THROOP, T. A. Total-Order Multi-Agent Task-Network Planning for Contract Bridge. En *AAAI/IAAI, Vol. 1*, páginas 108–113. 1996.

[128] SMITH, S. J. J., NAU, D. S. y THROOP, T. A. Computer Bridge - A Big Win for AI Planning. *AI Magazine*, vol. 19(2), páginas 93–106, 1998.

[129] SÁNCHEZ-RUIZ, A. A., GONZÁLEZ-CALERO, P. A. y DÍAZ-AGUDO, B. Planning with Description Logics and Syntactic Updates. En *Planning, Scheduling and Constraint Satisfaction (CAEPIA 2007 Workshop)* (editado por M. A. Salido y J. Fdez-Olivares), páginas 140–150. Universidad de Salamanca, 2007.

[130] SÁNCHEZ-RUIZ, A. A., GONZÁLEZ-CALERO, P. A. y DÍAZ-AGUDO, B. Abstraction in Knowledge-Rich Models for Case-Based Planning. En *ICCBR* (editado por L. McGinty y D. C. Wilson), vol. 5650 de *Lecture Notes in Computer Science*, páginas 313–327. Springer, 2009. ISBN 978-3-642-02997-4.

[131] SÁNCHEZ-RUIZ, A. A., GÓMEZ-MARTÍN, P. P., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Adaptation through Planning in Knowledge Intensive CBR. En [3], páginas 503–517.

[132] SÁNCHEZ-RUIZ, A. A., LEE-URBAN, S., MUÑOZ-AVILA, H., GONZÁLEZ-CALERO, P. A. y DÍAZ-AGUDO, B. Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval. En *Proceedings of the ICAPS-07 Workshop on ICAPS 2007 Workshop on Planning in Games*. 2007.

[133] SÁNCHEZ-RUIZ, A. A., LLANSÓ, D., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Authoring Behaviour for Characters in Games Reusing Abstracted Plan Traces. En *IVA* (editado por Z. Ruttkay, M. Kipp, A. Nijholt y H. H. Vilhjálmsson), vol. 5773 de *Lecture Notes in Computer Science*, páginas 56–62. Springer, 2009. ISBN 978-3-642-04379-6.

[134] SÁNCHEZ-RUIZ, A. A., LLANSÓ, D., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Authoring Behaviours for Game Characters Reusing Automatically Generated Abstract Cases. En *ICCBR Workshop* (editado por S. J. Delany), páginas 129–137. 2009.

[135] SÁNCHEZ-RUIZ, A. A., RECIO-GARCÍA, J. A., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Case Structures in jCOLIBRI. *Expert Update*, vol. 9(2), 2006. ISSN 1465-4091.

[136] SÁNCHEZ-RUIZ, A. A., RECIO-GARCÍA, J. A., GONZÁLEZ-CALERO, P. A. y DÍAZ-AGUDO, B. Towards Semi-Automatic Composition of CBR Systems in jCOLIBRI. En *Proccedings of the 11th UK Workshop on Case Based Reasoning* (editado por M. Petridis), páginas 48–59. CMS Press,University of Greenwich, 2006. ISBN 987-1-904521-39-6.

[137] TATE, A. Generating Project Networks. En *IJCAI*, páginas 888–893. 1977.

[138] TATE, A., DRABBLE, B. y KIRBY, R. O-Plan2: an Open Architecture for Command, Planning and Control. En *Intelligent Scheduling*, páginas 213–239. Morgan Kaufmann, 1994.

[139] THIEBAUX, S., HOFFMANN, J. y NEBEL, B. In Defense of PDDL Axioms. En *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, páginas 961–966. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[140] TSARKOV, D. y HORROCKS, I. FaCT++ Description Logic Reasoner: System Description. *Lecture Notes in Computer Science*, vol. 4130 LNAI, páginas 292–297, 2006.

[141] VELOSO, M. M. y CARBONELL, J. G. Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization. *Mach. Learn.*, vol. 10(3), páginas 249–278, 1993. ISSN 0885-6125.

[142] W3C. OWL 2 Web Ontology Language Document Overview.`http://www.w3.org/TR/owl2-overview/`. (Visitado en Abril de 2010).

[143] WEST, M. Evolve Your Hiearchy. *Game Developer*, vol. 13(3), páginas 51–54, 2006.

[144] WILKINS, D. y DESJARDINS, M. A Call for Knowledge-Based Planning. *AI Magazine*, vol. 22(1), 2001. ISSN 0738-4602.

[145] WILKINS, D. E. Can AI planners solve practical problems? *Computational Intelligence*, vol. 6, páginas 232–246, 1990.

[146] WINSLETT, M. Sometimes Updates are Circumscription. En *IJCAI'89: Proceedings of the 11th international joint conference on Artificial intelligence*, páginas 859–863. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[147] YANG, Q. Formalizing Planning Knowledge for Hierarchical Planning. *Computational Intelligence*, vol. 6, páginas 12–24, 1990.